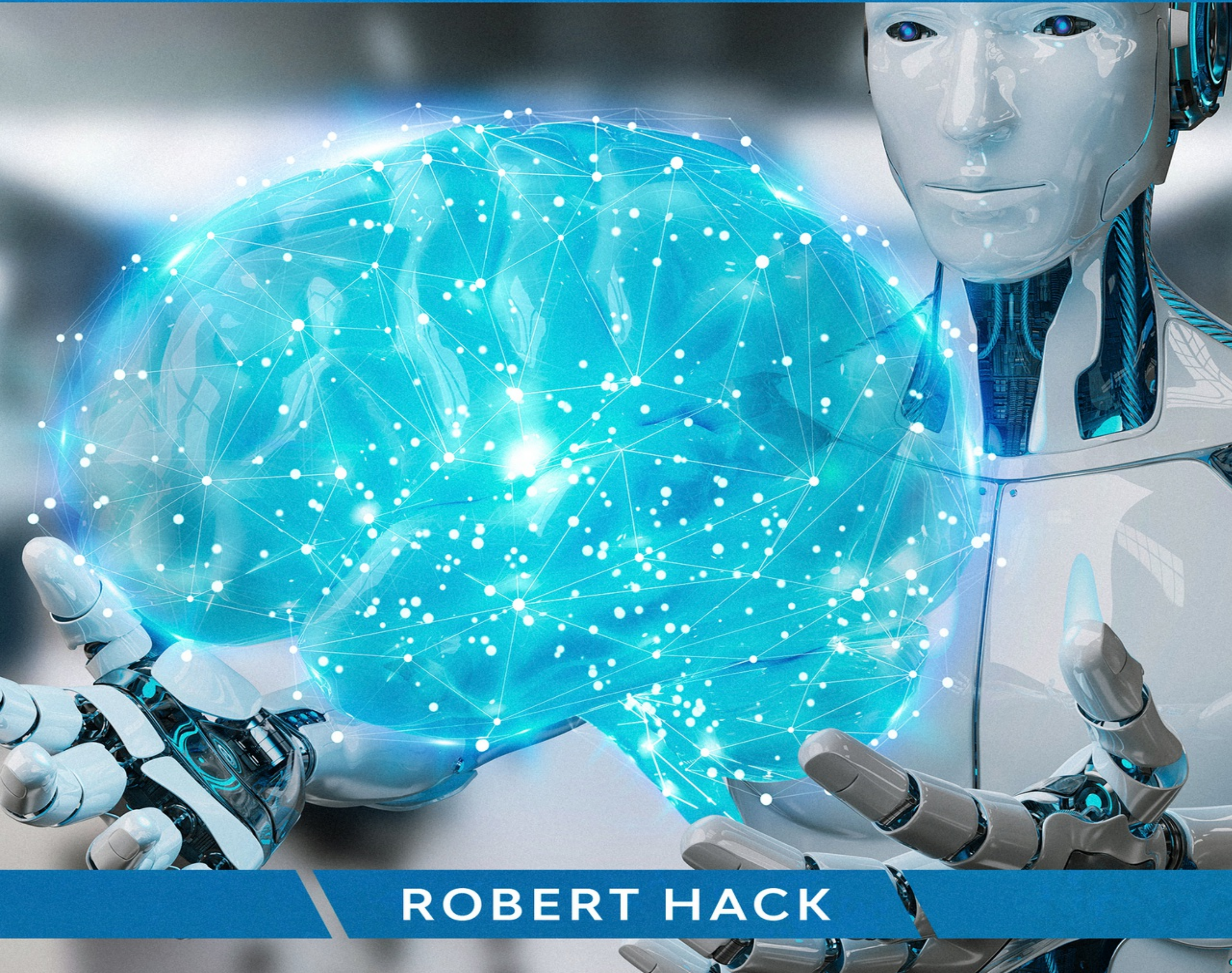
DEEP LEARNING

THE ULTIMATE BEGINNER'S GUIDE TO ARTIFICIAL INTELLIGENCE AND NEURAL NETWORKS. INTERMEDIATE, ADVANCED AND EXPERT CONCEPTS AND TECHNIQUES.



ROBERT HACK

DEEP LEARNING

THE ULTIMATE BEGINNER'S GUIDE TO ARTIFICIAL INTELLIGENCE AND NEURAL NETWORKS. INTERMEDIATE, ADVANCED AND EXPERT CONCEPTS AND TECHNIQUES.

AUTHOR

ROBERT HACK

This document is geared towards providing exact and reliable information with regards to the topic and issue covered. The publication is sold with the idea that the publisher is not required to render accounting, officially permitted, or otherwise, qualified services. If advice is necessary, legal or professional, a practiced individual in the profession should be ordered.

- From a Declaration of Principles which was accepted and approved equally by a Committee of the

The information provided herein is stated to be truthful and consistent, in that any liability, in terms of inattention or otherwise, by any usage or abuse of any policies, processes, or directions contained within is the solitary and utter responsibility of the recipient reader. Under no circumstances will any legal responsibility or blame be held against the publisher for any reparation, damages, or monetary loss due to the information herein, either directly or indirectly.

The information herein is offered for informational purposes solely, and is universal as so. The presentation of the information is without contract or any type of guarantee assurance.
The trademarks that are used are without any consent, and the publication of the trademark is without permission or backing by the trademark owner. All trademarks and brands within this book are for clarifying purposes only and are the owned by the owners themselves, not affiliated with this document

TABLE OF CONTENTS

INTRODUCTION TO DEEP LEARNING

You have probably heard a lot about deep learning. The term appears all over the place and seems to apply to everything. In reality, deep learning is a subset of machine learning, which in turn is a subset of artificial intelligence (AI). The first goal of this chapter is to help you understand what deep learning is really all about and how it applies to the world today. You may be surprised to learn that deep learning isn’t the only game in town; other methods of analyzing data exist. In fact, deep learning meets a specific set of needs when it comes to data analysis, so you might be using other methods and not even know it.

Deep learning is just a subset of AI, but it’s an important subset. You see deep learning techniques used for a number of tasks, but not every task. In fact, some people associate deep learning with tasks that it can’t perform. The next step in discovering deep learning is to understand what it can and can’t do for you.

As part of working with deep learning in this book, you write applications that rely on deep learning to process data and then produce a desired output. Of course, you need to know a little about the programming environment before you can do much. Yes, deep learning can perform amazing tasks when used appropriately, but it can also cause serious problems when applied to problems that it doesn’t support well. Sometimes you need to look to other technologies to perform a given task, or figure out which technologies to use with deep learning to provide a more efficient and elegant solution to specific problems.

Defining What Deep Learning Means

An understanding of deep learning begins with a precise definition of terms. Otherwise, you have a hard time separating the media hype from the realities of what deep learning can actually provide. Deep learning is part of both AI and machine learning, as shown in Figure 1-1. To understand deep learning, you must begin at the outside — that is, you start with AI, and then work your way through machine learning, and then finally define deep learning. The following sections help you through this process.

Starting from Artificial Intelligence

Saying that AI is an artificial intelligence doesn’t really tell you anything meaningful, which is why so many discussions and disagreements arise over this term. Yes, you can argue that what occurs is artificial, not having come from a natural source. However, the intelligence part is, at best, ambiguous. People define intelligence in many different ways. However, you can say that intelligence involves certain mental exercises composed of the following activities: » Learning: Having the ability to obtain and process new information. » Reasoning: Being able to manipulate information in various ways. » Understanding: Considering the result of information manipulation. » Grasping truths: Determining the validity of the manipulated information. » Seeing relationships: Divining how validated data interacts with other data. » Considering meanings: Applying truths to particular situations in a manner consistent with their relationship. » Separating fact from belief: Determining whether the data is adequately supported by provable sources that can be demonstrated to be consistently valid.

The list could easily get quite long, but even this list is relatively prone to interpretation by anyone who accepts it as viable. As you can see from the list, however, intelligence often follows a process that a computer system can mimic as part of a simulation:

1. Set a goal based on needs or wants.

2. Assess the value of any currently known information in support of the goal.

3. Gather additional information that could support the goal.

4. Manipulate the data such that it achieves a form consistent with existing information.

5. Define the relationships and truth values between existing and new information.

6. Determine whether the goal is achieved.

7. Modify the goal in light of the new data and its effect on the probability of success.

8. Repeat Steps 2 through 7 as needed until the goal is achieved (found true) or the possibilities for achieving it are exhausted (found false).

Even though you can create algorithms and provide access to data in support of this process within a computer, a computer's capability to achieve intelligence is severely limited. For example, a computer is incapable of understanding anything because it relies on machine processes to manipulate data using pure math in a strictly mechanical fashion. Likewise, computers can't easily separate truth from mistruth. In fact, no computer can fully implement any of the mental activities described in the list that describes intelligence.

When thinking about AI, you must consider the goals of the people who develop an AI. The goal is to mimic human intelligence, not replicate it. A computer doesn't truly think, but it gives the appearance of thinking. However, a computer actually provides this appearance only in the logical/mathematical form of intelligence. A computer is moderately successful in mimicking visual-spatial and bodily- kinesthetic intelligence. A computer has a low, passable capability in interpersonal and linguistic intelligence. Unlike humans, however, a computer has no way to mimic intrapersonal or creative intelligence.

Considering the role of AI

As described in the previous section, the first concept that's important to understand is that AI doesn't really have anything to do with human intelligence. Yes, some AI is modeled to simulate human intelligence, but that's what it is: a simulation. When thinking about AI, notice that an interplay exists between goal seeking, data processing used to achieve that goal, and data acquisition used to better understand the goal. AI relies on algorithms to achieve a result that may or may not have anything to do with human goals or methods of achieving those goals. With this in mind, you can categorize AI in four ways: » Acting humanly: When a computer acts like a human, it best reflects the Turing test, in which the computer succeeds when differentiation between the computer and a human isn't possible (see http://www.turing.org.uk/ scrapbook/test.html for details). This category also reflects what the media would have you believe that AI is all about. You see it employed for technologies such as natural language processing, knowledge representation, automated reasoning, and machine learning (all four of which must be present to pass the test). The original Turing Test didn't include any physical contact. The newer, Total Turing Test does include physical contact in the form of perceptual ability interrogation, which means that the computer must also employ both computer vision and robotics to succeed. Modern techniques include the idea of achieving the goal rather than mimicking humans completely. For example, the Wright brothers didn’t succeed in creating an airplane by precisely copying the flight of birds; rather, the birds provided ideas that led to aerodynamics, which in turn eventually led to human flight. The goal is to fly. Both birds and humans achieve this goal, but they use different approaches. » Thinking humanly: When a computer thinks as a human, it performs tasks that require intelligence (as contrasted with rote procedures) from a human to succeed, such as driving a car. To determine whether a program thinks like a human, you must have some method of determining how humans think, which the cognitive modeling approach defines. This model relies on three techniques:

• Introspection: Detecting and documenting the techniques used to achieve goals by monitoring one’s own thought processes.

• Psychological testing: Observing a person’s behavior and adding it to a database of similar behaviors from other persons given a similar set of circumstances, goals, resources, and environmental conditions (among other things).

• Brain imaging: Monitoring brain activity directly through various mechanical means, such as Computerized Axial Tomography (CAT), Positron Emission Tomography (PET), Magnetic Resonance Imaging (MRI), and Magnetoencephalography (MEG). After creating a model, you can write a program that simulates the model. Given the amount of variability among human thought processes and the difficulty of accurately representing these thought processes as part of a program, the results are experimental at best. This category of thinking humanly is often used in psychology and other fields in which modeling the human thought process to create realistic simulations is essential.

» Thinking rationally: Studying how humans think using some standard enables the creation of guidelines that describe typical human behaviors. A person is considered rational when following these behaviors within certain levels of deviation. A computer that thinks rationally relies on the recorded behaviors to create a guide as to how to interact with an environment based on the data at hand. The goal of this approach is to solve problems logically, when possible. In many cases, this approach would enable the creation of a baseline technique for solving a problem, which would then be modified to actually solve the problem. In other words, the solving of a problem in principle is often different from solving it in practice, but you still need a starting point. » Acting rationally: Studying how humans act in given situations under specific constraints enables you to determine which techniques are both efficient and effective. A computer that acts rationally relies on the recorded actions to interact with an environment based on conditions, environmental factors, and existing data. As with rational thought, rational acts depend on a solution in principle, which may not prove useful in practice. However, rational acts do provide a baseline upon which a computer can begin negotiating the successful completion of a goal. You find AI used in a great many applications today. The only problem is that the technology works so well that you don’t even know it exists. In fact, you might be surprised to find that many devices in your home already make use of this technology. The uses for AI number in the millions — all safely out of sight even when they’re quite dramatic in nature. Here are just a few of the ways in which you might see AI used: » Fraud detection: You get a call from your credit card company asking whether you made a particular purchase. The credit card company isn’t being nosy; it’s simply alerting you to the fact that someone else could be making a purchase using your card. The AI embedded within the credit card company’s code detected an unfamiliar spending pattern and alerted someone to it.

» Resource scheduling: Many organizations need to schedule the use of resources efficiently. For example, a hospital may have to determine where to put a patient based on the patient’s needs, availability of skilled experts, and the amount of time the doctor expects the patient to be in the hospital.

» Complex analysis: Humans often need help with complex analysis because there are literally too many factors to consider. For example, the same set of symptoms could indicate more than one problem. A doctor or other expert might need help making a diagnosis in a timely manner to save a patient’s life.

» Automation: Any form of automation can benefit from the addition of AI to handle unexpected changes or events. A problem with some types of automation today is that an unexpected event, such as an object in the wrong place, can actually cause the automation to stop. Adding AI to the automation can allow the automation to handle unexpected events and continue as though nothing happened.

» Customer service: The customer service line you call today may not even have a human behind it. The automation is good enough to follow scripts and use various resources to handle the vast majority of your questions. With good voice inflection (provided by AI as well), you may not even be able to tell that you're talking with a computer. » Safety systems: Many of the safety systems found in machines of various sorts today rely on AI to take over the vehicle in a time of crisis. For example, many automatic braking systems rely on AI to stop the car based on all the inputs that a vehicle can provide, such as the direction of a skid.

» Machine efficiency: AI can help control a machine in such a manner as to obtain maximum efficiency. The AI controls the use of resources so that the system doesn't overshoot speed or other goals. Every ounce of power is used precisely as needed to provide the desired services.

Focusing on machine learning

Machine learning is one of a number of subsets of AI. In machine learning, the goal is to create a simulation of human learning so that an application can adapt to uncertain or unexpected conditions. To perform this task, machine learning relies on algorithms to analyze huge datasets.

Currently, machine learning can't provide the sort of AI that the movies present (a machine can't intuitively learn as a human can); it can only simulate specific kinds of learning, and only in a narrow range at that. Even the best algorithms can't think, feel, present any form of self-awareness, or exercise free will. Characteristics that are basic to humans are frustratingly difficult for machines to grasp because of these limits in perception. Machines aren't self-aware.

What machine learning can do is perform predictive analytics far faster than any human can. As a result, machine learning can help humans work more efficiently. The current state of AI, then, is one of performing analysis, but humans must still consider the implications of that analysis: making the required moral and ethical decisions. The essence of the matter is that machine learning provides just the learning part of AI, and that part is nowhere near ready to create an AI of the sort you see in films.

The main point of confusion between learning and intelligence is that people assume that simply because a machine gets better at its job (it can learn), it’s also aware (has intelligence). Nothing supports this view of machine learning. The same phenomenon occurs when people assume that a computer is purposely causing problems for them. The computer can’t assign emotions and therefore acts only upon the input provided and the instruction contained within an application to process that input. A true AI will eventually occur when computers can finally emulate the clever combination used by nature: » Genetics: Slow learning from one generation to the next » Teaching: Fast learning from organized sources » Exploration: Spontaneous learning through media and interactions with others

To keep machine learning concepts in line with what the machine can actually do, you need to consider specific machine learning uses. It’s useful to view uses of machine learning outside the normal realm of what many consider the domain of AI.

Here are a few uses for machine learning that you might not associate with an AI:

» Access control: In many cases, access control is a yes-or-no proposition. An employee smartcard grants access to a resource in much the same way as people have used keys for centuries. Some locks do offer the capability to set times and dates that access is allowed, but such coarse-grained control doesn’t really answer every need. By using machine learning, you can determine whether an employee should gain access to a resource based on role and need. For example, an employee can gain access to a training room when the training reflects an employee role.

» Animal protection: The ocean might seem large enough to allow animals and ships to cohabitate without problem. Unfortunately, many animals get hit by ships each year. A machine learning algorithm could allow ships to avoid animals by learning the sounds and characteristics of both the animal and the ship. (The ship would rely on underwater listening gear to track the animals through their sounds, which you can actually hear a long distance from the ship.)

» Predicting wait times: Most people don’t like waiting when they have no idea of how long the wait will be. Machine learning allows an application to determine waiting times based on staffing levels, staffing load, complexity of the problems the staff is trying to solve, availability of resources, and so on.

Moving from machine learning to deep learning

Deep learning is a subset of machine learning, as previously mentioned. In both cases, algorithms appear to learn by analyzing huge amounts of data (however, learning can occur even with tiny datasets in some cases). However, deep learning varies in the depth of its analysis and the kind of automation it provides.

You can summarize the differences between the two like this:

» A completely different paradigm: Machine learning is a set of many different techniques that enable a computer to learn from data and to use what it learns to provide an answer, often in the form of a prediction. Machine learning relies on different paradigms such as using statistical analysis, finding analogies in data, using logic, and working with symbols. Contrast the myriad techniques used by machine learning with the single technique used by deep learning, which mimics human brain functionality. It processes data using computing units, called neurons, arranged into ordered sections, called layers. The technique at the foundation of deep learning is the neural network.

» Flexible architectures: Machine learning solutions offer many knobs (adjustments) called hyperparameters that you tune to optimize algorithm learning from data. Deep learning solutions use hyperparameters, too, but they also use multiple user-configured layers (the user specifies number and type). In fact, depending on the resulting neural network, the number of layers can be quite large and form unique neural networks capable of specialized learning: Some can learn to recognize images, while others can detect and parse voice commands. The point is that the term deep is appropriate; it refers to the large number of layers potentially used for analysis. The architecture consists of the ensemble of different neurons and their arrangement in layers in a deep learning solution.

» Autonomous feature definition: Machine learning solutions require human intervention to succeed. To process data correctly, analysts and scientist use a lot of their own knowledge to develop working algorithms. For instance, in a machine learning solution that determines the value of a house by relying on data containing the wall measures of different rooms, the machine learning algorithm won’t be able to calculate the surface of the house unless the analyst specifies how to calculate it beforehand. Creating the right information for a machine learning algorithm is called feature creation, which is a time consuming activity. Deep learning doesn’t require humans to perform any feature-creation activity because, thanks to its many layers, it defines its own best features. That’s also why deep learning outperforms machine learning in otherwise very difficult tasks such as recognizing voice and images, understanding text, or beating a human champion at the Go game (the digital form of the board game in which you capture your opponent’s territory).

You need to understand a number of issues with regard to deep learning solutions, the most important of which is that the computer still doesn’t understand anything and isn’t aware of the solution it has provided. It simply provides a form of feedback loop and automation conjoined to produce desirable outputs in less time than a human could manually produce precisely the same result by manipulating a machine learning solution. The second issue is that some benighted people have insisted that the deep learning layers are hidden and not accessible to analysis. This isn’t the case. Anything a computer can build is ultimately traceable by a human. In fact, the General Data Protection Regulation (GDPR) requires that humans perform such analysis. The requirement to perform this analysis is controversial, but current law says that someone must do it.

The third issue is that self-adjustment goes only so far. Deep learning doesn’t always ensure a reliable or correct result. In fact, deep learning solutions can go horribly wrong. Even when the application code doesn’t go wrong, the devices used to support the deep learning can. Even so, with these problems in mind, you can see deep learning used for a number of extremely popular applications.

Using Deep Learning in the Real World

Make no mistake: People do use deep learning in the real world to perform a broad range of tasks. For example, many automobiles today use a voice interface. The voice interface can perform basic tasks, even right from the outset. However, the more you talk to it, the better the voice interface performs. The interface learns as you talk to it — not only the manner in which you say things, but also your personal preferences. The following sections give you a little information on how deep learning works in the real world.

Understanding the concept of learning

When humans learn, they rely on more than just data. Humans have intuition, along with an uncanny grasp of what will and what won’t work. Part of this inborn knowledge is instinct, which is passed from generation to generation through DNA. The way humans interact with input is also different from what a computer will do. When dealing with a computer, learning is a matter of building a database consisting of a neural network that has weights and biases built into it to ensure proper data processing. The neural network then processes data, but not in a manner that’s even remotely the same as what a human will do.

Performing deep learning tasks

Humans and computers are best at different tasks. Humans are best at reasoning, thinking through ethical solutions, and being emotional. A computer is meant to process data — lots of data — really fast. You commonly use deep learning to solve problems that require looking for patterns in huge amounts of data — problems whose solution is non intuitive and not immediately noticeable. The article at http://www.yaronhadad.com/deep-learning-most-amazing-applications/ tells you about 30 different ways in which people are currently using deep learning to perform tasks. In just about every case, you can sum up the problem and its solution as processing huge amounts of data quickly, looking for patterns, and then relying on those patterns to discover something new or to create a particular kind of output.

Employing deep learning in applications

Deep learning can be a stand-alone solution, as illustrated in this book, but it’s often used as part of a much larger solution and mixed with other technologies. For example, mixing deep learning with expert systems is not uncommon. However, real applications are more than just numbers generated from some nebulous source. When working in the real world, you must also consider various kinds of data sources and understand how those data sources work. A camera may require a different sort of deep learning solution to obtain information from it, while a thermometer or proximity detector may output simple numbers (or analog data that requires some sort of processing to use). Real-world solutions are messy, so you need to be prepared with more than one solution to problems in your toolkit.

Considering the Deep Learning Programming Environment

You may automatically assume that you must jump through a horrid set of hoops and learn esoteric programming skills to delve into deep learning. It’s true that you gain flexibility by writing applications using one of the programming languages that work well for deep learning needs. However, Deep Learning and other products like it are enabling people to create deep learning solutions without programming. Essentially, such solutions involve describing what you want as output by defining a model graphically. These kinds of solutions work well for straightforward problems that others have already had to solve, but they lack the flexibility to do something completely different — a task that requires something more than simple analysis.

Deep learning solutions in the cloud, such as that provided by Amazon Web Services (AWS) can give you additional flexibility. These environments also tend to make the development environment simpler by providing as much or little support as you want. In fact, AWS provides support for various kinds of serverless computing (https://aws. amazon.com/serverless/) in which you don’t worry about any sort of infrastructure. However, these solutions can become quite expensive. Even though they give you greater flexibility than using a premade solution, they still aren’t as flexible as using an actual development environment.

You have other non-programming solutions to consider as well. For example, if you want power and flexibility, but don’t want to program to get it, you could rely on a product such as MATLAB, which provide a deep learning toolkit. MATLAB and certain other environments do focus more on the algorithms you want to use, but to gain full functionality from them, you need to write scripts as a minimum, which means that you’re dipping your toe into programming to some extent. A problem with these environments is that they can also be lacking in the power department, so some solutions may take longer than you expect.

At some point, no matter how many other solutions you try, serious deep learning problems will require programming. When reviewing the choices online, you often see AI, machine learning, and deep learning all lumped together. However, just as the three technologies work at different levels, so do the programming languages that you require. A good deep learning solution will require the use of multiprocessing, preferably using a Graphics Processing Unit (GPU) with lots of cores. Your language of choice must also support the GPU through a compatible library or package. So, just choosing a language usually isn't enough; you need to investigate further to ensure that the language will actually meet your needs. With this caution in mind, here are the top languages (in order of popularity, as of this writing) for deep learning use :

» Python » R » MATLAB (the scripting language, not the product) » Octave The only problem with this list is that other developers have other opinions. Python and R normally appear at the top of everyone's lists, but after that you can find all sorts of opinions. When choosing a language, you usually have to consider these issues:

» Learning curve: Your experiences have a lot to say about what you find easiest to learn. Python is probably the best choice for someone who has programmed for a number of years, but R might be the better choice for someone who has already experienced functional programming. MATLAB or Octave might work best for a math professional.

» Speed: Any sort of deep learning solution will require a lot of processing power. Many people say that because R is a statistical language, it offers more in the way of statistical support and usually provides a faster result. Actually, Python's support for great parallel programming probably offsets this advantage when you have the required hardware.

» Community support: Many forms of community support exist, but the two that are most important for deep learning are help in defining a solution and access to a wealth of premade programming aids. Of the four, Octave probably provides the least in the way of community support; Python provides the most.

» Cost: How much a language costs depends on the kind of solution you choose and where you run it. For example, MATLAB is a proprietary product that requires purchase, so you have something invested immediately when using MATLAB. However, even though the other languages are free at the outset, you can find hidden costs, such as running your code in the cloud to gain access to GPU support. » DNN Frameworks support: A framework can make working with your language significantly easier. However, you have to have a framework that works well with all other parts of your solution. The two most popular frameworks are TensorFlow and PyTorch. Oddly enough, Python is the only language that supports both, so it offers you the greatest flexibility. You use Caffe with MATLAB and TensorFlow with R.

» Production ready: A language has to support the kind of output needed for your project. In this regard, Python shines because it's a general-purpose language. You can create any sort of application needed with it. However, the more specific environments provided by the other languages can be incredibly helpful with some projects, so you need to consider all of them.

Overcoming the Deep Learning Hype

Previous parts of this chapter discuss some issues with the perception of deep learning, such as some people's belief that it appears everywhere and does everything. The problem with deep learning is that it has been a victim of its own media campaign. Deep learning solves specific sorts of problems. The following sections help you avoid the hype associated with deep learning.

Discovering the start-up ecosystem

Using a deep learning solution is a lot different from creating a deep learning solution of your own. The infographic at https://www.analyticsvidhya.com/blog/2018/08/infographic-complete-deep-learning-path/ gives you some ideas on how to get started with Python (a process this book simplifies for you). The educational requirements alone can take a while to fulfill. However, after you have worked through a few projects on your own, you begin to realize that the hype surrounding deep learning extends all the way to the start of setup. Deep learning isn't a mature technology, so trying to use it is akin to building a village on the moon or deep diving the Marianas Trench. You're going to encounter issues, and the technology will constantly change on you.

Some of the methods used to create deep learning solutions need work, too. The concept of a computer actually learning anything is false, as is the idea that computers have any form of sentience at all. The reason that Microsoft, Amazon, and other vendors have problems with deep learning is that even their engineers have unrealistic expectations. Deep learning comes down to math and pattern matching — really fancy math and pattern matching, to be sure, but the idea that it's anything else is simply wrong.

Knowing when not to use deep learning

Deep learning is only one way to perform analysis, and it’s not always the best way. For example, even though expert systems are considered old technology, you can’t really create a self-driving car without one for the reasons described at https://aitrends.com/ai-insider/expert-systems-ai-self-driving-cars- crucial-innovative-techniques/. A deep learning solution turns out to be way too slow for this particular need. Your car will likely contain a deep learning solution, but you’re more likely to use it as part of the voice interface. AI in general and deep learning in particular can make the headlines when the technology fails to live up to expectations. For example, the article at https:// www.techrepublic.com/article/top-10-ai-failures-of-2016/ provides a list of AI failures, some of which relied on deep learning as well. It’s a mistake to think that deep learning can somehow make ethical decisions or that it will choose the right course of action based on feelings (which no machine has). Anthropomorphizing the use of deep learning will always be a mistake. Some tasks simply require a human.

Speed and the capability to think like a human are the top issues for deep learning, but there are many more. For example, you can’t use deep learning if you don’t have sufficient data to train it. In fact, the article at https://www.sas.com/en_us/ insights/articles/big-data/5-machine-learning-mistakes.html offers a list of five common mistakes that people make when getting into machine learning and deep learning environments. If you don’t have the right resources, deep learning will never work.

CONCEPTUAL FOUNDATIONS

Introducing the Machine Learning Principles

As discussed in chapter 1, the concept of learning for a computer is different from the concept of learning for humans. However, Chapter 1 doesn't really describe machine learning, the kind of learning a computer uses, in any depth. After all, what you're really looking at is an entirely different sort of learning that some people would view as a combination of math, pattern matching, and data storage. This chapter begins by pointing the way to a deeper understanding of how machine learning works.

However, an explanation of machine learning doesn't completely help you understand what's going on when you work with it. How machine learning works is also important, which is the subject of the next section of the chapter. In this section, you discover that no perfect methods exist for performing analysis. You may have to experiment with your analysis to get the expected output. In addition, different approaches to machine learning are available, and each has advantages and disadvantages.

The third part of the chapter takes what you've discovered in the previous two parts and helps you apply it. No matter how you shape your data and perform analysis on it, machine learning is the wrong approach in some cases and will never provide you with useful output. Knowing the right uses for machine learning is essential if you want to receive consistent output that helps you perform interesting tasks. The whole purpose of machine learning is to learn something interesting from the data and then to do something interesting with it.

Defining Machine Learning

Here's a short definition of machine learning: It's an application of AI that can automatically learn and improve from experience without being explicitly programmed to do so. The learning occurs as a result of analyzing ever increasing amounts of data, so the basic algorithms don't change, but the code's internal weights and biases used to select a particular answer do. Of course, nothing is quite this simple. The following sections discuss more about what machine learning is so that you can understand its place within the world of AI and what deep learning acquires from it.

Data scientists often refer to the technology used to implement machine learning as algorithms. An algorithm is a series of step-by-step operations, usually computations that can solve a defined problem in a finite number of steps. In machine learning, the algorithms use a series of finite steps to solve the problem by learning from data.

Understanding how machine learning works

Machine learning algorithms learn, but it’s often hard to find a precise meaning for the term learning because different ways exist to extract information from data, depending on how the machine learning algorithm is built. Generally, the learning process requires huge amounts of data that provides an expected response given particular inputs. Each input/response pair represents an example and more examples make it easier for the algorithm to learn. That’s because each input/response pair fits within a line, cluster, or other statistical representation that defines a problem domain. Learning is the act of optimizing a model, which is a mathematical, summarized representation of data itself, such that it can predict or otherwise determine an appropriate response even when it receives input that it hasn’t seen before. The more accurately the model can come up with correct responses, the better the model has learned from the data inputs provided. An algorithm fits the model to the data, and this fitting process is training.

In this case, starting with input values of 1, 4, 5, 8, and 10 and pairing them with their corresponding outputs of 7, 13, 15, 21, and 25, the machine learning algorithm determines that the best way to represent the relationship between the input and output is the formula 2x + 5. This formula defines the model used to process the input data — even new, unseen data —to calculate a corresponding output value. The trend line (the model) shows the pattern formed by this algorithm, such that a new input of 3 will produce a predicted output of 11. Even though most machine learning scenarios are much more complicated than this (and the algorithm can’t create rules that accurately map every input to a precise output), the example gives provides you a basic idea of what happens. Rather than have to individually program a response for an input of 3, the model can compute the correct response based on input/response pairs that it has learned.

Understanding that it’s pure math

The central idea behind machine learning is that you can represent reality by using a mathematical function that the algorithm doesn’t know in advance, but which it can guess after seeing some data (always in the form of paired inputs and outputs). You can express reality and all its challenging complexity in terms of unknown mathematical functions that machine learning algorithms find and make available as a modification of their internal mathematical function. That is, every machine learning algorithm is built around a modifiable math function. The function can be modified because it has internal parameters or weights for such a purpose. As a result, the algorithm can tailor the function to specific information taken from data. This concept is the core idea for all kinds of machine learning algorithms.

Learning in machine learning is purely mathematical, and it ends by associating certain inputs with certain outputs. It has nothing to do with understanding what the algorithm has learned. (When humans analyze data, we build an understanding of the data to a certain extent.) The learning process is often described as training because the algorithm is trained to match the correct answer (the output) to every question offered (the input). (Machine Learning For Dummies, by John Paul Mueller and Luca Massaron, [Wiley], describes how this process works in detail.)

In spite of lacking deliberate understanding and of being a mathematical process, machine learning can prove useful in many tasks. It provides many AI applications the power to mimic rational thinking given a certain context when learning occurs by using the right data.

Learning by different strategies

Machine learning offers a number of different ways to learn from data. Depending on your expected output and on the type of input you provide, you can categorize algorithms by learning style. The style you choose depends on the sort of data you have and the result you expect. The four learning styles used to create algorithms are » Supervised » Unsupervised » Self-supervised » Reinforcement

The following sections discuss learning styles.

Supervised

When working with supervised algorithms, the input data is labeled and has a specific expected result. You use training to create a model that an algorithm fits to the data. As training progresses, the predictions or classifications become more accurate. Here are some examples of supervised learning algorithms: » Linear or Logistic regression » Support Vector Machines (SVMs) » Naïve Bayes » K-Nearest Neighbors (KNN)

You need to distinguish between regression problems, whose target is a numeric value, and classification problems, whose target is a qualitative variable, such as a class or tag. A regression task could determine the average prices of houses in the Boston area, while an example of a classification task is distinguishing between kinds of iris flowers based on their sepal and petal measures. Here are some examples of supervised learning:

Data Input (X) Data Output (y) Real-World Application

History of customers' purchases

A list of products that customers have never bought

Recommender system

Images A list of boxes labeled with an object name

Image detection and recognition

English text in the form of questions

English text in the form of answers

Chatbot, a software application that can converse

English text German text Machine language translation

Audio Text transcript Speech recognition

Image, sensor data Steering, braking, or accelerating

Behavioral planning for autonomous driving

Unsupervised When working with unsupervised algorithms, the input data isn't labeled and the results aren't known. In this case, analysis of structures in the data produces the required model. The structural analysis can have a number of goals, such as to reduce redundancy or to group similar data. Examples of unsupervised learning are

» Clustering

» Anomaly detection

» Neural networks

Self-Supervised

You’ll find all sorts of kinds of learning described online, but self-supervised learning is in a category of its own. Some people describe it as autonomous supervised learning, which gives you the benefits of supervised learning but without all the work required to label data.

Theoretically, self-supervised could solve issues with other kinds of learning that you may currently use. The following list compares self-supervised learning with other sorts of learning that people use. » Supervised learning: The closest form of learning associated with self- supervised learning is supervised learning because both kinds of learning rely on pairs of inputs and labeled outputs. In addition, both forms of learning are associated with regression and classification. However, the difference is that self-supervised learning doesn’t require a person to label the output. Instead, it relies on correlations, embedded metadata, or domain knowledge embedded within the input data to contextually discover the output label.

» Unsupervised learning: Like unsupervised learning, self-supervised learning requires no data labeling. However, unsupervised learning focuses on data structure — that is, patterns within the data. Therefore, you don’t use self-supervised learning for tasks such as clustering, grouping, dimensionality reduction, recommendation engines, or the like.

» Semi-supervised learning: A semi-supervised learning solution works like an unsupervised learning solution in that it looks for data patterns. However, semi-supervised learning relies on a mix of labeled and unlabeled data to perform its tasks faster than is possible using strictly unlabeled data. Self Supervised learning never requires labels and uses context to perform its task, so it would actually ignore the labels when supplied.

Reinforcement

You can view reinforcement learning as an extension of self-supervised learning because both forms use the same approach to learning with unlabeled data to achieve similar goals. However, reinforcement learning adds a feedback loop to the mix. When a reinforcement learning solution performs a task correctly, it receives positive feedback, which strengthens the model in connecting the target inputs and output. Likewise, it can receive negative feedback for incorrect solutions. In some respects, the system works much the same as working with a dog based on a system of rewards.

Training, validating, and testing data

Machine learning is a process, just as everything is a process in the world of computers. To build a successful machine learning solution, you perform these tasks as needed, and as often as needed: » Training: Machine learning begins when you train a model using a particular algorithm against specific data. The training data is separate from any other data, but it must also be representative. If the training data doesn't truly represent the problem domain, the resulting model can't provide useful results. During the training process, you see how the model responds to the training data and make changes, as needed, to the algorithms you use and the manner in which you massage the data prior to input to the algorithm. » Validating: Many datasets are large enough to split into a training part and a testing part. You first train the model using the training data, and then you validate it using the testing data. Of course, the testing data must again represent the problem domain accurately. It must also be statistically compatible with the training data. Otherwise, you won't see results that reflect how the model will actually work. » Testing: After a model is trained and validated, you still need to test it using real-world data. This step is important because you need to verify that the model will actually work on a larger dataset that you haven't used for either training or testing. As with the training and validation steps, any data you use during this step must reflect the problem domain you want to interact with using the machine learning model.

Training provides a machine learning algorithm with all sorts of examples of the desired inputs and outputs expected from those inputs. The machine learning algorithm then uses this input to create a math function. In other words, training is the process whereby the algorithm works out how to tailor a function to the data. The output of such a function is typically the probability of a certain output or simply a numeric value as output.

To give an idea of what happens in the training process, imagine a child learning to distinguish trees from objects, animals, and people. Before the child can do so in an independent fashion, a teacher presents the child with a certain number of tree images, complete with all the facts that make a tree distinguishable from other objects of the world. Such facts could be features, such as the tree's material (wood), its parts (trunk, branches, leaves or needles, roots), and location (planted in the soil). The child builds an understanding of what a tree looks like by contrasting the display of tree features with the images of other, different examples, such as pieces of furniture that are made of wood, but do not share other characteristics with a tree.

A machine learning classifier works the same. A classifier algorithm provides you with a class as output. For instance, it could tell you that the photo you provide as an input matches the tree class (and not an animal or a person). To do so, it builds its cognitive capabilities by creating a mathematical formulation that includes all the given input features in a way that creates a function that can distinguish one class from another.

Looking for generalization

To be useful, a machine learning model must represent a general view of the data provided. If the model doesn’t follow the data closely enough, it’s underfitted — that is, not fitted enough because of a lack of training. On the other hand, if the model follows the data too closely, it’s overfitted, following the data points like a glove because of too much training. Underfitting and overfitting both cause problems because the model isn’t generalized enough to produce useful results. Given unknown input data, the resulting predictions or classifications will contain large error values. Only when the model is correctly fitted to the data will it provide results within a reasonable error range.

This whole issue of generalization is also important in deciding when to use machine learning. A machine learning solution always generalizes from specific examples to general examples of the same sort. How it performs this task depends on the orientation of the machine learning solution and the algorithms used to make it work.

The problem for data scientists and others using machine learning and deep learning techniques is that the computer won’t display a sign telling you that the model correctly fits the data. Often, it’s a matter of human intuition to decide when a model is trained enough to provide a good generalized result. In addition, the solution creator must choose the right algorithm out of the thousands that exist. Without the right algorithm to fit the model to the data, the results will be disappointing. To make the selection process work, the data scientist must possess

» A strong knowledge of the available algorithms

» Experience dealing with the kind of data in question » An understanding of the desired output

» A desire to experiment with various algorithms

The last requirement is the most important because there are no hard-and-fast rules that say a particular algorithm will work with every kind of data in every possible situation. If this were the case, so many algorithms wouldn't be available. To find the best algorithm, the data scientist often resorts to experimenting with a number of algorithms and comparing the results.

Getting to know the limits of bias

Your computer has no bias. It has no goal of world domination or of making your life difficult. In fact, computers don't have goals of any kind. The only thing a computer can provide is output based on inputs and processing technique. However, bias still gets into the computer and taints the results it provides in a number of ways: » Data: The data itself can contain mistruths or simply misrepresentations. For example, if a particular value appears twice as often in the data as it does in the real world, the output from a machine learning solution is tainted, even though the data itself is correct.

» Algorithm: Using the wrong algorithm will cause the machine learning solution to fit the model to the data incorrectly.

» Training: Too much or too little training changes how the model fits the data and therefore the result.

» Human interpretation: Even when a machine learning solution outputs a correct result, the human using that output can misinterpret it. The results are every bit as bad as, and perhaps worse than, when the machine learning solution fails to work as anticipated.

You need to consider the effects of bias no matter what sort of machine learning solution you create. It's important to know what sorts of limits these biases place on your solution and whether the solution is reliable enough to provide useful output.

Keeping model complexity in mind

Simpler is always better when it comes to machine learning. Many different algorithms may provide you with useful output from your machine learning solution, but the best algorithm to use is the one that's easiest to understand and provides the most straightforward results. Occam's Razor is generally recognized as the best strategy to follow. Basically, Occam's Razor tells you to use the simplest solution that will solve a particular problem. As complexity increases, so does the potential for errors.

Considering the Many Different Roads to Learning

The learning part of machine learning makes it dynamic — that is, able to change itself when it receives additional data. The capability to learn makes machine learning different from other sorts of AI, such as knowledge graphs and expert systems. It doesn't make machine learning better than other AI (as described in Chapter 1), but simply useful for a certain set of problems. Of course, the problem with quantifying what learning entails is that humans and computers view learning differently. In addition, computers use different learning techniques than humans do and some humans may not see the learning part of machine learning as learning at all. The following sections discuss the methods that machine learning algorithms use to learn so that you can better understand that machine learning and human learning are inherently different.

Understanding there is no free lunch

You may have heard the common myth that you can have everything in the way of computer output without putting much effort into deriving the solution. Unfortunately, no absolute solution exists to any problem, and better answers are often quite costly. When working with algorithms, you quickly discover that some algorithms perform better than others in solving certain problems, but that there also isn't a single algorithm that works best on every problem. This is because of the math behind algorithms. Certain math functions are good at representing some problems but may hit a wall on certain other problems. Each algorithm has its specialty.

Discovering the five main approaches

Algorithms come in various forms and perform various tasks. One way to categorize algorithms is by school of thought — the method that a group of like minded thinkers believed would solve a particular kind of problem. Of course, other ways to categorize algorithms exist, but this approach has the advantage of helping you understand algorithm uses and orientations better. The following sections provide an overview of the five main algorithmic techniques.

Symbolic reasoning A group called the symbologists relies on algorithms that use symbolic reasoning to find a solution to problems. The term inverse deduction commonly appears as induction. In symbolic reasoning, deduction expands the realm of human knowledge, while induction raises the level of human knowledge. Induction commonly opens new fields of exploration, and deduction explores those fields. However, the most important consideration is that induction is the science portion of this type of reasoning, while deduction is the engineering. The two strategies work hand in hand to solve problems by first opening a field of potential exploration to solve the problem and then exploring that field to determine whether it does, in fact, solve it.

As an example of this strategy, deduction would say that if a tree is green and that green trees are alive, the tree must be alive. When thinking about induction, you would say that the tree is green and that the tree is also alive; therefore, green trees are alive. Induction provides the answer to what knowledge is missing given a known input and output.

Neural networks

Neural networks are the brainchild of a group called the connectionists. This group of algorithms strives to reproduce the brain’s functions using silicon instead of neurons. Essentially, each of the neurons (created as an algorithm that models the real-world counterpart) solves a small piece of the problem, and the use of many neurons in parallel solves the problem as a whole.

A neural network can provide a method of correction for errant data, and the most popular of these methods is backpropagation. The use of backpropagation, or backward propagation of errors, seeks to determine the conditions under which errors are removed from networks built to resemble the human neurons by changing the weights (how much a particular input figures into the result) and biases (which features are selected) of the network. The goal is to continue changing the weights and biases until such time as the actual output matches the target output.

At this point, the artificial neuron fires and passes its solution along to the next neuron in line. The solution created by each individual neuron is only part of the whole solution. Each neuron continues to pass information to the next neuron in line until the group of neurons creates a final output.

Evolutionary algorithms

A group called the evolutionaries relies on the principles of evolution to solve problems. This strategy is based on the survival of the fittest, removing any solutions that don't match the desired output. A fitness function determines the viability of each function in solving a problem.

Using a tree structure, the solution method looks for the best solution based on function output. The winner of each level of evolution gets to build the next level of functions. The next level will get closer to solving the problem but may not solve it completely, which means that another level is needed. This particular algorithmic type relies heavily on recursion and languages that strongly support recursion to solve problems. An interesting output of this strategy has been algorithms that evolve themselves: One generation of algorithms actually builds the next generation.

Bayesian inference

The Bayesians use various statistical methods to solve problems. Given that statistical methods can create more than one apparently correct solution, the choice of a function becomes one of determining which function has the highest probability of succeeding. For example, when using these techniques, you might accept a set of symptoms as input. An algorithm will compute the probability that a particular disease will result from the symptoms as output. Given that multiple diseases have the same symptoms, the probability is important because a user will see some situations in which a lower probability output is actually the correct output for a given circumstance.

Ultimately, Bayesian algorithms rely on the idea of never quite trusting any hypothesis (a result that someone has given you) completely without seeing the evidence used to make it (the input the other person used to make the hypothesis). Analyzing the evidence proves or disproves the hypothesis that it supports. Consequently, you can't determine which disease someone has until you test all the symptoms. One of the most recognizable outputs from this group of algorithms is the spam filter.

Systems that learn by analogy

The analogizers use kernel machines to recognize patterns in data. By recognizing the pattern of one set of inputs and comparing it to the pattern of a known output, you can create a problem solution. The goal is to use similarity to determine the best solution to a problem. It's the kind of reasoning that determines that using a particular solution worked in a given circumstance at some previous time; therefore, using that solution for a similar set of circumstances should also work. One of the most recognizable outputs from this group of algorithms is recommender systems. For example, when you get on Amazon and buy a product, the recommender system comes up with other, related, products that you might also want to buy.

Delving into some different approaches

It helps to have several views of algorithms so that you understand what they do and why they do it. The previous section looks at algorithms based on the groups that created them. However, you have other approaches you can use to categorize algorithms. The following list categorizes some popular algorithms by similarity: » Artificial neural network: Models the structure or function of biological neural networks (or sometimes it does both). The goal is to perform pattern matching for regression and classification problems. However, the technique mimics the approach used by biological organisms rather than strictly relying on a true math-based approach.

Here are examples of artificial neural network algorithms:

- Perceptron

- Feed-forward Neural Network

- Hopfield Network

- Radial Basis Function Network (RBFN)

- Self-Organizing Map (SOM)

» Association rule: Extracts rules that help explain the relationships between variables in data. You can use these rules to discover useful associations within huge datasets that are ordinarily easy to miss.

Here are the more popular association-rule algorithms:

- Apriori algorithm

- Eclat algorithm

» Bayesian: Applies Bayes' Theorem to probability problems. This form of algorithm sees use for classification and regression problems.

Here are examples of Bayesian algorithms:

- Naïve Bayes

- Gaussian Naïve Bayes

- Multinomial Naïve Bayes

- Bayesian Belief Network (BBN)

- Bayesian Network (BN)

» Clustering: Describes a model for organizing data by class or other criteria. The results are often centroid or hierarchical in nature. What you see are data relationships in a way that helps make sense of the data — that is, how the values affect each other.

The following list contains examples of clustering algorithms:

- K-means

- K-medians

- Expectation Maximisation (EM)

- Hierarchical Clustering

» Decision tree: Constructs a model of decisions based on the actual values found in data. The resulting tree structure enables you to perform comparisons between new data and existing data very quickly. This form of algorithm often sees use for classification and regression problems.

The following list shows some of the common decision-tree algorithms: • Classification and Regression Tree (CART)

- Iterative Dichotomiser 3 (ID3)

- C4.5 and C5.0 (different versions of a powerful approach)

- Chi-squared Automatic Interaction Detection (CHAID)

» Deep learning: Provides an update to artificial neural networks that rely on multiple layers to exploit even larger datasets and build complex neural networks. This particular group of algorithms works well with semi supervised learning problems in which the amount of labeled data is minimal.

Here are some examples of deep learning algorithms:

• Deep Boltzmann Machine (DBM)

• Deep Belief Networks (DBN)

• Convolutional Neural Network (CNN)

• Recurrent Neural Network (RNN)

• Stacked Auto-Encoders

» Dimensionality reduction: Seeks and exploits similarities in the structure of data in a manner similar to clustering algorithms, but using unsupervised methods. The purpose is to summarize or describe data using less information so that the dataset becomes smaller and easier to manage. In some cases, people use these algorithms for classification or regression problems. Here is a list of common dimensionality reduction algorithms: • Principal Component Analysis (PCA) • Factor Analysis (FA) • Multidimensional Scaling (MDS) • t-Distributed Stochastic Neighbor Embedding (t-SNE) » Ensemble: Composes a group of multiple weaker models into a cohesive whole whose individual predictions are combined in some manner to define an overall prediction. Using an ensemble can solve certain problems faster, more efficiently, or with reduced errors. Here are some common ensemble algorithms: • Boosting • Bootstrapped Aggregation (Bagging) • AdaBoost • Random Forest • Gradient Boosting Machines (GBM) » Instance-based: Defines a model for decision problems in which the training data consists of examples that are later used for comparison purposes. A similarity measure helps determine when new examples compare favorably to existing examples within the database. Some people call these algorithms winner-take-all or memory-based learning because of the manner in which they work. The following list provides some common algorithms associated with this category: • K-Nearest Neighbors (KNN) • Learning Vector Quantization (LVQ) » Regression: Models the relationship among variables. This relationship is iteratively refined using an error measure. This category sees heavy use in statistical machine learning. The following list shows the algorithms normally associated with this kind of algorithm: • Ordinary Least Squares Regression (OLSR) • Logistic Regression » Regularization: Regulates other algorithms by penalizing complex solutions and favoring simpler ones. This kind of algorithm often sees use with regression methods. The goal is to ensure that the solution doesn't become lost in its own complexity and delivers solutions within a given time frame using the least number of resources. Here are examples of regularization algorithms: • Ridge Regression • Least Absolute Shrinkage and Selection Operator (LASSO) • Elastic Net • Least-Angle Regression (LARS) » Support Vector Machines (SVM): Supervised learning algorithms that solve classification and regression problems by separating only a few data examples (called supports, hence the name of the algorithm) from the rest of the data using a function. After separating these supports, the prediction becomes easier. The form of analysis depends on the function type (called a kernel): linear, polynomial, or radial basis. Here are examples of SVM algorithms. • Linear Support Vector Machines • Radial Basis Function Support Vector Machines • One-Class Support Vector Machines (for unsupervised learning) » Other: You have many other algorithms from which to choose. This list contains major algorithm categories. Some of the categories not found in this list belong to those used for feature selection, algorithm accuracy, performance measures, and specialty subfields of machine learning. For example, whole categories of algorithms are devoted to the topic of Computer Vision (CV) and Natural Language Processing (NLP). As you read through this book, you find many other categories of algorithms and may begin to wonder how a data scientist can make any choice, much less the right one.

Awaiting the next breakthrough

Breakthroughs require patience because computers are inherently based on math. You may not see them as such when working with a higher-level language like Python, but everything that goes on beneath the hood requires an extreme understanding of math and the data it manipulates. Consequently, you can expect to see new uses for machine learning and deep learning in the future as scientists continue to find new ways to process data, create algorithms, and use those algorithms to define data models.

Unfortunately, working with what is available today isn't enough to create the applications of tomorrow (despite what the movies might have you believe). In the future, you can expect advances in hardware to make applications that aren't feasible today quite doable. It's not just a matter of additional computing power or larger memories. Tomorrow's computer will have access to sensors that aren't available today; processors that do things that today’s processors can’t; and methods of viewing how computers think that haven’t been envisioned yet. What the world needs most now is experience, and experience always takes time to accumulate.

Pondering the True Uses of Machine Learning

The fact that you have a number of options to choose from when it comes to AI means that machine learning isn’t the only technology you should consider to solve any given problem. Machine learning does excel at helping you solve specific categories of problems. To determine where machine learning works best, you must begin by considering how an algorithm learns and then applying that knowledge to problem classes that you need to solve. Remember that machine learning is about generalization, so it doesn’t work particularly well in these scenarios: » The result must provide a precise answer, such as calculating a trip to Mars. » You can solve the problem using generalization but other techniques are simpler, such as developing software to compute a factorial of a number. » You don’t have a good generalization of the problem because the problem is misunderstood, no specific relationship exists between inputs and results, or the problem domain is too complex.

The following sections discuss the true uses of machine learning from the perspective of how it learns and then defines the benefits of machine learning given specific problem domains.

Understanding machine learning benefits

How you can benefit from machine learning depends partly on your environment and partly on what you expect from it. For example, if you spend time on Amazon buying products, you might expect machine learning to make useful recommendations based on past purchases at some point. These recommendations are for products that you might not have otherwise known about. Recommending products that you already use or don’t need isn’t particularly useful, which is where the machine learning part comes into play. As Amazon builds more data about your purchasing habits, the recommendations should become more useful, although not even the best machine learning algorithm will ever guess your needs correctly every time.

Of course, machine learning benefits you in many other ways. A developer can use machine learning to add an NLP capability to an application. A researcher could use it to help find the next cure for cancer. You may already use it for spam filtering for your e-mail or rely on it when you get into your car as part of a voice interface. With this in mind, the following benefits likely fit more of a business perspective for using machine learning effectively, but keep in mind that many other ways exist as well: » Simplify product marketing: One of the issues that any organization faces is determining what to sell and when, based on customer preferences. Sales campaigns are expensive, so having one fail usually isn't an option. In addition, an organization might find odd bits of information: Customers may like products in red but not in green. Knowing what the customer wants is incredibly difficult unless you can analyze huge amounts of buying data, which is something that machine learning does well. » Predict future sales accurately: Being in business can seem a little like gambling because you can't be quite sure that your bets will pay off. A machine learning solution can follow sales minute by minute and track trends before they become obvious. The capability to perform this kind of tracking means that you can more accurately tune sales channels to deliver optimal results and ensure that stores have enough of the right products to sell. It isn't precisely like gazing into a crystal ball, but it's close.

» Forecast medical and other employee downtime: Oddly enough, some organizations end up having problems because employees choose the worst possible times to be absent from work. In some cases, these absences seem unpredictable, such as medical needs, while in others you could possibly predict them, such as a sudden need for personal time. By tracking various trends from easily available data sources, you can track both medical-type and personal-type absences for your industry as a whole, location as a whole, and your organization in particular to ensure that you have enough people to get the job done at any given time.

» Reduce data entry errors: Some kinds of data entry errors are relatively easy to avoid by using form features correctly or incorporating a spell checker into your application. In addition, adding certain kinds of pattern matching can help reduce capitalization errors or incorrect phone numbers. Machine learning can take error reduction to another level by correctly identifying complex patterns that other techniques will miss. For example, a customer order may need one of part A and two of part B to create a whole unit. The pattern matching for these kinds of sales can be elusive, but machine learning can make it possible, reducing errors that are particularly different to find and eradicate.

» Improve financial rule and modeling precision: Keeping the finances straight can prove difficult in an organization of any size. Machine learning enables you to perform tasks such as portfolio management, algorithmic trading, loan underwriting, and fraud detection with greater precision. You can't eliminate human participating in such cases, but the human and machine working together can become an incredibly efficient combination that won't allow many errors to pass unnoticed.

» Foresee maintenance needs: Any system that consists of something physical likely requires maintenance of various sorts. For example, machine learning can help predict when a system will need cleaning based on past performance and environmental monitoring. You can also do things like plan for replacement or repair of certain equipment based on past repairs and equipment statistics. A machine learning solution can even enable you to determine whether replacement or repair is the better option.

» Augment customer interaction and improve satisfaction: Customers like to feel special; in fact, everyone does. However, trying to create a custom plan for each customer manually would prove impossible. You can find a wealth of information about customers through online sources, including everything from recent purchases to consistent buying habits. By combining all this data with a good machine learning solution and customer support personnel who have discerning eyes, you can appear to have personally created a special solution for each customer, even though the time required to do so is minimal.

Discovering machine learning limits

The limits of a technology are often hard to quantify completely because these limits are often the result of a lack of imagination on the part of the creator or consumer of that technology. However, machine learning does have some distinct limits that you need to consider before using this technology to perform any given task. The following list isn't complete. In fact, you may not even completely agree with it, but it does provide a good starting point.

» Massive amounts of training data are needed: Unlike programmed solutions of the past, a machine learning solution relies on massive amounts of data to train it. As problem complexity increases, the number of data points required to model a particular problem increases, making even more data necessary. Although humans generate increasingly larger amounts of data in specific problem domains and the computing power needed to process this data also increases daily, some problem domains simply lack enough data or enough processing power to make machine learning effective. » Labeling data is tedious and error prone: When using the supervised learning technique (see the "Learning by different strategies" section, earlier in this chapter, for details), someone must label the data to provide the output value. The labeling process for huge amounts of data is both tedious and time consuming, making machine learning difficult at times. The problem is that a human can look at any number of examples of something like a stop sign and know that they're all stop signs, but a computer must have every stop sign individually labelled.

» Machines can't explain themselves: As machine learning solutions become more flexible and capable; the amount of hidden functionality becomes greater as well. In fact, when dealing with deep learning solutions, you find that the solution contains one or usually more hidden layers that the solution creates but that humans haven't taken the time to explore. Consequently, both machine learning (to some extent) and deep learning (to a greater extent) encounter issues for which transparency is valued and counter to some laws, such as the General Data Protection Regulation, or GDPR (https://eugdpr.org/). Because the process becomes opaque, a human must now analyze a process that is supposed to be automatic. A potential solution for this problem may come in the form of new strategies, such as Local Interpretable Model-Agnostic Explanations (LIME) (see https://homes.cs.washington.edu/~marcotcr/ blog/lime/ for details).

» Bias makes the results less usable: An algorithm can't tell when data contains various mistruths in it (Artificial Intelligence For Dummies, by John Paul Mueller and Luca Massaron [Wiley], explains this issue in detail). Consequently, it regards all data as being unbiased and completely truthful. As a result, any analysis performed by an algorithm trained using this data is suspect. The problem becomes even greater when the algorithm itself is biased. You can find countless examples online of algorithms misidentifying common objects like stop signs because of the combination of data containing mistruths and biased algorithms.

» Machine learning solutions can’t cooperate: One of the most important advantages of being human is the ability to collaborate with others. Knowledge potential increases exponentially as each party to a potential solution submits its piece of knowledge to create a whole that is much greater than the sum of its parts. A single machine learning solution remains a single machine learning solution because of it can’t generalize knowledge and thereby contribute to a comprehensive solution with multiple cooperative parties

NEURAL NETWORKS: THE BUILDING BLOCKS OF DEEP LEARNING

INTRODUCING NEURAL NETWORKS

You may have heard the term neural network in reference to artificial intelligence. The first thing you need to know is that the correct term is Artificial Neural Network (ANN) because no one has discovered any method of recreating a real brain, which is where the concept of a neural network comes from. Chapter 2 of this book describes the various approaches to deep learning, of which ANNs are one. You find the term shortened in this book because everyone else is using the short term, but you need to know that ANN is actually the correct term and that they're the work of the connectionist tribe.

After you get past the whole idea that your computer lacks a brain — at least a real brain — you can begin to appreciate the perceptron, which is the simplest type of neural network. The perceptron is the focus of many of the neural network pictures you see online, but not all neural networks mimic the perceptron.

A neural network can work with complex data because of how it allows multiple inputs to flow through multiple layers of processing to produce myriad outputs. (The perceptron can only actually choose between two outputs.) The idea is that each of the paths fires only when it actually has a chance of answering whatever question you pose with your inputs, based on the algorithms you choose. The next section of the chapter discusses some of these methods of dealing with complex data.

Because neural networks can model incredibly complex data in a manner that amazes some people, you might think it can correct for errors in processing, such as overfitting. Unfortunately, computers really don't have real brains, so overfitting is a problem that you need to solve. The final section of this chapter looks at some solutions for overfitting and discusses why it's such a big problem in the first place.

Discovering the Incredible Perceptron

Even though this book is about deep learning, you still need to know something about the previous implementation levels of machine learning and AI. The perceptron is actually a type (implementation) of machine learning for most people, but other sources will tell you that it’s a true form of deep learning. You can start the journey toward discovering how machine learning algorithms work by looking at models that figure out their answers using lines and surfaces to divide examples into classes or to estimate value predictions. These are linear models, and this chapter presents one of the earliest linear algorithms used in machine learning: the perceptron. Later chapters will help you discover other sorts of modeling significantly more advanced than the perceptron. However, before you can advance to these other topics, you should understand the interesting history of the perceptron.

Understanding perceptron functionality

Frank Rosenblatt, of the Cornell Aeronautical Laboratory, devised the perceptron in 1957 under the sponsorship of the United States Naval Research. Rosenblatt was a psychologist and pioneer in the field of artificial intelligence. Proficient in cognitive science, his idea was to create a computer that could learn by trial and error, just as a human does.

The idea was successfully developed, and at the beginning, the perceptron wasn’t conceived as just a piece of software; it was created as software running on dedicated hardware. You can see it at https://blogs.umass.edu/comphon/2017/ 06/15/did-frank-rosenblatt-invent-deep-learning-in-1962/. Using that combination allowed faster and more precise recognition of complex images than any other computer could do at the time. The new technology raised great expectations and caused a huge controversy when Rosenblatt affirmed that the perceptron was the embryo of a new kind of computer that would be able to walk, talk, see, write, and even reproduce itself and be conscious of its existence. If true, it would have been a powerful tool, and it introduced the world to AI.

Needless to say, the perceptron didn’t realize the expectations of its creator. It soon displayed a limited capacity, even in its image-recognition specialization. The general disappointment ignited the first AI winter (a period of reduced funding and interest due to overhyping, for the most part) and the temporary abandonment of connectionism until the 1980s.

Connectionism is the approach to machine learning that is based on neuroscience as well as the example of biologically interconnected networks. You can retrace the root of connectionism to the perceptron.

The perceptron is an iterative algorithm that strives to determine, by successive and reiterative approximations, the best set of values for a vector, w, which is also called the coefficient vector. When the perceptron has achieved a suitable coefficient vector, it can predict whether an example is part of a class. For instance, one of the tasks the perceptron initially performed was to determine whether an image received from visual sensors resembled a boat (an image recognition example required by the United States Office of Naval Research, the sponsor of the research on the perceptron). When the perceptron saw the image as part of the boat class, it meant that it classified the image as a boat.

Vector w can help predict the class of an example when you multiply it by the matrix of features, X, containing the information in numeric values expressed in numeric values relative to your example, and then add the result of the multiplication to a constant term, called the bias, b. If the result of the sum is zero or positive, perceptron classifies the example as part of the class. When the sum is negative, the example isn't part of the class. Here's the perceptron formula, in which the sign function outputs 1 (when the example is part of the class) when the value inside the parenthesis is equal or above zero; otherwise, it outputs 0:

$y = \text{sign}(Xw + b)$

Note that this algorithm contains all the elements that characterize a deep neural network, meaning that all the building blocks enabling the technology were present since the beginning: » Numeric processing of the input: X contains numbers, and no symbolic values are used as input until you process it as a number. For instance, you can't input symbolic information such as red, green, or blue until you convert these color values to numbers. » Weights and bias: The perceptron transforms X by multiplying by the weights and adding the bias. » Summation of results: Uses matrix multiplication when multiplying X by the w vector (an aspect of matrix multiplication covered in Chapter 5). » Activation function: The perceptron activates a result of the input being part of the class when the summation exceeds a threshold — in this case, when the resulting sum is zero or more. » Iterative learning of the best set of values for the vector w: The solution relies on successive approximations based on the comparison between the perceptron output and the expected result.

Touching the non separability limit

The secret to perceptron calculations is in how the algorithm updates the vector w values. Such updates happen by randomly picking one of the misclassified examples. You have a misclassified example when the perceptron determines that an example is part of the class, but it isn’t, or when the perceptron determines an example isn’t part of the class, but it is. The perceptron handles one misclassified example at a time (call it xt) and operates by changing the w vector using a simple weighted addition:

w = w + ŋ(xt * yt)

This formula is called the update strategy of the perceptron, and the letters stand for different numerical elements: » The letter w is the coefficient vectors, which is updated to correctly show whether the misclassified example t is part of the class or not. » The Greek letter eta (η) is the learning rate. It’s a floating number between 0 and 1. When you set this value near zero, it can limit the capability of the formula to update the vector w almost completely, whereas setting the value near one makes the update process fully impact the w vector values. Setting different learning rates can speed up or slow down the learning process. Many other algorithms use this strategy, and lower eta is used to improve the optimization process by reducing the number of sudden w value jumps after an update. The trade-off is that you have to wait longer before getting the concluding results. » The xt variable refers to the vector of numeric features for the example t. » The yt variable refers to the ground truth of whether the example t is part of the class or not. For the perceptron algorithm, yt is numerically expressed with +1 when the example is part of the class and with –1 when the example is not part of the class.

The update strategy provides intuition about what happens when using a perceptron to learn the classes. If you imagine the examples projected on a Cartesian plane, the perceptron is nothing more than a line trying to separate the positive class from the negative one. As you may recall from linear algebra, everything expressed in the form of y = xb+a is actually a line in a plane. The perceptron uses a formula of y = xw + b, which uses different letters but expresses the same form, the line in a Cartesian plane.

Initially, when w is set to zero or to random values, the separating line is just one of the infinite possible lines found on a plane. The updating phase defines it by forcing it to become nearer to the misclassified point. As the algorithm passes through the misclassified examples, it applies a series of corrections. In the end, using multiple iterations to define the errors, the algorithm places the separating line at the exact border between the two classes.

In spite of being such a smart algorithm, the perceptron showed its limits quite soon. Apart from being capable of guessing two classes using only quantitative features, it had an important limit: If two classes had no border because of mixing, the algorithm couldn’t find a solution and kept updating itself infinitely.

If you can’t divide two classes spread on two or more dimensions by any line or plane, they’re nonlinearly separable. Overcoming data’s being nonlinearly separable is one of the challenges that machine learning has to overcome in order to become effective against complex problems based on real data, not just on artificial data created for academic purposes. When the nonlinearly separability matter came under scrutiny and practitioners started losing interest in the perceptron, experts quickly theorized that they could fix the problem by creating a new feature space in which previously inseparable classes are tuned to become separable. Thus, the perceptron would be as fine as before. Unfortunately, creating new feature spaces is a challenge because it requires computational power that’s only partially available to the public today. Creating a new feature space is an advanced topic discussed later in the book when studying the learning strategies of algorithms, such as neural networks and support vector machines.

In recent years, the algorithm has had a revival thanks to big data: the perceptron, in fact, doesn’t need to work with all the data in memory, but it can do fine using single examples (updating its coefficient vector only when a misclassified case makes it necessary). It’s therefore a perfect algorithm for online learning, such as learning from big data an example at a time.

Hitting Complexity with Neural Networks

The previous section of the chapter helped you discover the neural network from the perspective of the perceptron. Of course, there is more to neural networks than that simple beginning. The capacity and other issues that plague the perceptron see at least partial resolution in newer algorithms. The following sections help you understand neural networks as they exist today.

Considering the neuron

The core neural network component is the neuron (also called a unit). Many neurons arranged in an interconnected structure make up a neural network, with each neuron linking to the inputs and outputs of other neurons. Thus, a neuron can input features from examples or the results of other neurons, depending on its location in the neural network.

When the psychologist Rosenblatt conceived the perceptron, he thought of it as a simplified mathematical version of a brain neuron. A perceptron takes values as inputs from the nearby environment (the dataset), weights them (as brain cells do, based on the strength of the in-bound connections), sums all the weighted values, and activates when the sum exceeds a threshold. This threshold outputs a value of 1; otherwise, its prediction is 0. Unfortunately, a perceptron can’t learn when the classes it tries to process aren’t linearly separable. However, scholars discovered that even though a single perceptron couldn’t learn the logical operation XOR (the exclusive or, which is true only when the inputs are dissimilar), two perceptrons working together could. Neurons in a neural network are a further evolution of the perceptron: they take many weighted values as inputs, sum them, and provide the summation as the result, just as a perceptron does. However, they also provide a more sophisticated transformation of the summation, something that the perceptron can’t do. In observing nature, scientists noticed that neurons receive signals but don’t always release a signal of their own. It depends on the amount of signal received. When a neuron acquires enough stimuli, it fires an answer; otherwise, it remains silent. In a similar fashion, algorithmic neurons, after receiving weighted values, sum them and use an activation function to evaluate the result, which transforms it in a nonlinear way. For instance, the activation function can release a zero value unless the input achieves a certain threshold, or it can dampen or enhance a value by non linearly rescaling it, thus transmitting a rescaled signal.

A neural network has different activation functions, The linear function (labeled Binary step) doesn’t apply any transformation, and it’s seldom used because it reduces a neural network to a regression with polynomial transformations. Neural networks commonly use the sigmoid (labeled Logistic) or the hyperbolic tangent (labeled TanH), or the ReLU (which is by far the more common today) activation functions. The figure shows how an input (expressed on the horizontal axis) can transform an output into something else (expressed on the vertical axis). The examples show a binary step, a logistic (also called sigmoid), and a tangent hyperbolic activation function (often referred to as tanh).

You learn more about activation functions later in the chapter, but note for now that activation functions clearly work well in certain ranges of x values. For this reason, you should always rescale inputs to a neural network using statistical standardization (zero mean and unit variance) or normalize the input in the range from 0 to 1 or from –1 to 1.

Activation functions are what make a neural network perform in a classification or regression; yet, the initial choice of the sigmoid or tanh activations for most networks pose a critical limit when using networks that are more complex, because both activations work optimally for a very restricted range of values.

Pushing data with feed-forward

In a neural network, you must consider the architecture, which is how the neural network components are arranged. Contrary to other algorithms, which have a fixed pipeline that determines how algorithms receive and process data, neural networks require you to decide how information flows by fixing the number of units (the neurons) and their distribution in layers.

The figure shows a simple neural architecture. Note how the layers filter information in a progressive way. This is a feed-forward input because data feeds one way forward into the network. Connections exclusively link the units in one layer with the units in the following layer (information flow from left to right). No connections exist between units in the same layer or with units outside the next layer. Moreover, the information pushes forward (from the left to the right). Processed data never returns to previous neuron layers.

Using a neural network is like using a stratified filtering system for water: You pour the water from above and the water is filtered at the bottom. The water has no way to go back; it just goes forward and straight down, and never laterally. In the same way, neural networks force data features to flow through the network and mix with each other only according to the network's architecture. By using the best architecture to mix features, the neural network creates new composed features at every layer and helps achieve better predictions. Unfortunately, there is no way to determine the best architecture without empirically trying different solutions and testing whether the output data helps predict your target values after flowing through the network.

The first and last layers play an important role. The first layer, called the input layer, picks ups the features from each data example processed by the network. The last layer, called the output layer, releases the results.

A neural network can process only numeric, continuous information; it can’t be constrained to work with qualitative variables (for example, labels indicating a quality such as red, blue, or green in an image). You can process qualitative variables by transforming them into a continuous numeric value, such as a series of binary values. When a neural network processes a binary variable, the neuron treats the variable as a generic number and turns the binary values into other values, even negative ones, by processing across units.

Note the limitation of dealing only with numeric values, because you can’t expect the last layer to output a nonnumeric label prediction. When dealing with a regression problem, the last layer is a single unit. Likewise, when you’re working with a classification and you have output that must choose from a number n of classes, you should have n terminal units, each one representing a score linked to the probability of the represented class. Therefore, when classifying a multiclass problem such as iris species, the final layer has as many units as species. For instance, in the archetypal Iris classification example, created by the famous statistician Fisher, you have three classes: setosa, versicolor, and virginica. In a neural network based on the Iris dataset, you therefore have three units representing one of the three Iris species. For each example, the predicted class is the one that gets the higher score at the end.

Some neural networks have special final layers, collectively called softmax, which can adjust the probability of each class based on the values received from a previous layer. In classification, the final layer may represent both a partition of probabilities thanks to softmax (a multiclass problem in which total probabilities sum to 100 percent) or an independent score prediction (because an example can have more classes, which is a multilabel problem in which summed probabilities can be more than 100 percent). When the classification problem is a binary classification, a single output suffices. Also, in regression, you can have multiple output units, each one representing a different regression problem. (For instance, in forecasting, you can have different predictions for the next day, week, month, and so on.)

Going even deeper into the rabbit hole

Neural networks have different layers, each one having its own weights. Because the neural network segregates computations by layers, knowing the reference layer is important because you can account for certain units and connections. You can refer to every layer using a specific number and generically talk about each layer using the letter l.

Each layer can have a different number of units, and the number of units located between two layers dictates the number of connections. By multiplying the number of units in the starting layer with the number in the following layer, you can determine the total number of connections between the two: number of connections(l) = units(l) * units(l+1).

A matrix of weights, usually named with the uppercase Greek letter Theta (θ), represents the connections. For ease of reading, the book uses the capital letter W, which is a fine choice because it is a matrix or a multi-dimensional array. Thus, you can use W1 to refer to the connection weights from layer 1 to layer 2, W2 for the connections from layer 2 to layer 3, and so on.

Weights represent the strength of the connection between neurons in the network. When the weight of the connection between two layers is small, it means that the network dumps values flowing between them and signals that taking this route won't likely influence the final prediction. Alternatively, a large positive or negative value affects the values that the next layer receives, thus changing certain predictions. This approach is analogous to brain cells, which don't stand alone but connect with other cells. As someone grows in experience, connections between neurons tend to weaken or strengthen to activate or deactivate certain brain network cell regions, causing other processing or an activity (a reaction to a danger, for instance, if the processed information signals a life-threatening situation) Now that you know some conventions regarding layers, units, and connections, you can start examining the operations that neural networks execute in detail. To begin, you can call inputs and outputs in different ways: » a: The result stored in a unit in the neural network after being processed by the activation function (called g). This is the final output that is sent further along the network. » z: The multiplication between a and the weights from the W matrix. z represents the signal going through the connections, analogous to water in pipes that flows at a higher or lower pressure depending on the pipe diameter. In the same way, the values received from the previous layer get higher or lower values because of the connection weights used to transmit them.

Each successive layer of units in a neural network progressively processes the values taken from the features (picture a conveyor belt). As data transmits in the network, it arrives at each unit as a value produced by the summation of the values present in the previous layer and weighted by connections represented in the matrix W. When the data with added bias exceeds a certain threshold, the activation function increases the value stored in the unit; otherwise, it extinguishes the signal by reducing it. After processing by the activation function, the result is ready to push forward to the connection linked to the next layer. These steps repeat for each layer until the values reach the end and you have a result.

The figure shows a detail of the process that involves two units pushing their results to another unit. This event happens in every part of the network. When you understand the passage from two neurons to one, you can understand the entire feed-forward process, even when more layers and neurons are involved. For more explanation, here are the seven steps used to produce a prediction in a neural network made of four layers:

1. The first layer (notice the superscript 1 on a) loads the value of each feature in a different unit:
2. 2. The weights of the connections bridging the input layer with the second layer are multiplied by the values of the units in the first layer. A matrix multiplication weights and sums the inputs for the second layer together.
3. $z(2)=W(1)a(1)$ 3. The algorithm adds a bias constant to layer two before running the activation function. The activation function transforms the second layer inputs. The resulting values are ready to pass to the connections.
4. $a(2) = g(z(2) + bias(2))$ 4. The third layer connections weigh and sum the outputs of layer two.
5. $z(3) = W(2)a(2)$ 5. The algorithm adds a bias constant to layer three before running the activation function. The activation function transforms the layer-three inputs.
6. $a(3) = g(z(3) + bias(3))$ 6. The layer-three outputs are weighted and

summed by the connections to the output layer.

7. z(4) = W(3)a(3) 7. Finally, the algorithm adds a bias constant to layer four before running the activation function. The output units receive their inputs and transform the input using the activation function. After this final transformation, the output units are ready to release the resulting predictions of the neural network.
8. a(4) = g(z(4) + bias(4))
9. The activation function plays the role of a signal filter, helping to select the relevant signals and avoid the weak and noisy ones (because it discards values below a certain threshold). Activation functions also provide nonlinearity to the output because they enhance or damp the values passing through them in a non proportional way.

   10. The weights of the connections provide a way to mix and compose the features in a new way, creating new features in a way not too different from a polynomial expansion. The activation renders nonlinear the resulting recombination of the features by the connections. Both of these neural network components enable the algorithm to learn complex target functions that represent the relationship between the input features and the target outcome.

Using backpropagation to adjust learning

From an architectural perspective, a neural network does a great job of mixing signals from examples and turning them into new features to achieve an approximation of complex nonlinear functions (functions that you can’t represent as a straight line in the features’ space). To create this capability, neural networks work as universal approximators, which means that they can guess any target function. However, you have to consider that one aspect of this feature is the capacity to model complex functions (representation capability), and another aspect is the capability to learn from data effectively. Learning occurs in a brain because of the formation and modification of synapses between neurons, based on stimuli received by trial- and-error experience. Neural networks provide a way to replicate this process as a mathematical formulation called backpropagation.

Since its early appearance in the 1970s, the backpropagation algorithm has been given many fixes. Each neural network learning process improvement resulted in new applications and a renewed interest in the technique. In addition, the current deep learning revolution, a revival of neural networks, which were abandoned at the beginning of the 1990s, is due to key advances in the way neural networks learn from their errors. As seen in other algorithms, the cost function activates the necessity to learn certain examples better (large errors correspond to high costs). When an example with a large error occurs, the cost function outputs a high value that is minimized by changing the parameters in the algorithm. The optimization algorithm determines the best action for reducing the high outputs from the cost function.

In linear regression, finding an update rule to apply to each parameter (the vector of beta coefficients) is straightforward. However, in a neural network, things are a bit more complicated. The architecture is variable and the parameter coefficients (the connections) relate to each other because the connections in a layer depend on how the connections in the previous layers recombined the inputs. The solution to this problem is the backpropagation algorithm. Backpropagation is a smart way to propagate the errors back into the network and make each connection adjust its weights accordingly. If you initially feed-forward propagated information to the network, it’s time to go backward and give feedback on what went wrong in the forward phase.

Backpropagation is how adjustments required by the optimization algorithm are propagated through the neural network. Distinguishing between optimization and backpropagation is important. In fact, all neural networks use backpropagation, but the next chapter discusses many different optimization algorithms. Discovering how backpropagation works isn’t complicated, even though demonstrating how it works using formulas and mathematics requires derivatives and the proving of some formulations, which is quite tricky and beyond the scope of this book. To get a sense of how backpropagation operates, start from the end of the network, just at the moment when an example has been processed and you have a prediction as an output. At this point, you can compare it with the real result and, by subtracting the two results, get an offset, which is the error. Now that you know the mismatch of the results at the output layer, you can progress backward in order to distribute it along all the units in the network.

The cost function of a neural network for classification is based on cross-entropy (as seen in logistic regression):

$$Cost = y * \log(hW(X)) + (1 - y)*\log(1 - hW(X))$$

This is a formulation involving logarithms. It refers to the prediction produced by the neural network and expressed as hW(X) (which reads as the result of the network given connections W and X as input). To make things easier, when thinking of the cost, it helps to think of it as computing the offset between the expected results and the neural network output.

The first step in transmitting the error back into the network relies on backward multiplication. Because the values fed to the output layer are made of the contributions of all units, proportional to the weight of their connections, you can redistribute the error according to each contribution. For instance, the vector of errors of a layer n in the network, a vector indicated by the Greek letter delta (δ), is the result of the following formulation:

$$\delta\ (n) = W(n)T * \delta\ (n+1)$$

This formula says that, starting from the final delta, you can continue redistributing delta going backward in the network and using the weights you used to push forward the value to partition the error to the different units. In this way, you can get the terminal error redistributed to each neural unit, and you can use it to recalculate a more appropriate weight for each network connection to minimize the error. To update the weights W of layer l, you just apply the following formula:

$$W(l) = W(1) + \eta * \delta\ (1) * g'(z(l))\ \ *a(1)$$

The formula may appear puzzling at first sight, but it is a summation, and you can discover how it works by looking at its elements. First, look at the function g’. It’s the first derivative of the activation function g, evaluated by the input values z. In fact, this is the gradient descent method. Gradient descent determines how to reduce the error measure by finding, among the possible combinations of values, the weights that most reduce the error.

The Greek letter eta (η), sometimes also called alpha (α) or epsilon (ε) depending on the textbook you consult, is the learning rate. As found in other algorithms, it reduces the effect of the update suggested by the gradient descent derivative. In fact, the direction provided may be only partially correct or just roughly correct. By taking multiple small steps in the descent, the algorithm can take a more precise direction toward the global minimum error, which is the target you want to achieve (that is, a neural network producing the least possible prediction error).

Different methods are available for setting the right eta value, because the optimization largely depends on it. One method sets the eta value starting high and reduces it during the optimization process. Another method variably increases or decreases eta based on the improvements obtained by the algorithm: large improvements call a larger eta (because the descent is easy and straight); smaller improvements call a smaller eta so that the optimization will move slower, looking for the best opportunities to descend. Think of it as being on a tortuous path in the mountains: You slow down and try not to be struck or thrown off the road as you descend.

Most implementations offer an automatic setting of the correct eta. You need to note this setting is relevance when training a neural network because it's one of the important parameters to tweak to obtain better predictions, together with the layer architecture. Weight updates can happen in different ways with respect to the training set of examples: » Online mode: The weight update happens after every example traverses the network. In this way, the algorithm treats the learning examples as a stream from which to learn in real time. This mode is perfect when you have to learn out of core, that is, when the training set can't fit into RAM memory. However, this method is sensitive to outliers, so you have to keep your learning rate low. (Consequently, the algorithm is slow to converge to a solution.) » Batch mode: The weight update happens after processing all the examples in the training set. This technique makes optimization fast and less subject to having variance appear in the example stream. In batch mode, the backpropagation considers the summed gradients of all examples. » Mini-batch (or stochastic) mode: The weight update happens after the network has processed a subsample of randomly selected training set examples. This approach mixes the advantages of online mode (low memory usage) and batch mode (a rapid convergence) while introducing a random element (the subsampling) to avoid having the gradient descent Struggling with Overfitting.

Given the neural network architecture, you can imagine how easily the algorithm could learn almost anything from data, especially if you added too many layers. In fact, the algorithm does so well that its predictions are often affected by a high estimate variance called overfitting. Overfitting causes the neural network to learn every detail of the training examples, which makes replicating them in the prediction phase possible. But, apart from the training set, the network won't ever correctly predict anything different. The following sections discuss some of the issues with overfitting in more detail.

Understanding the problem

When you use a neural network for a real problem, you become stricter and more cautious in an implementation than you do with other algorithms. Neural networks are frailer and more prone to relevant errors than other machine learning solutions.

You carefully split your data into training, validation, and test sets. Before the algorithm learns from data, you must evaluate the goodness of your parameters:

» Architecture (the number of layers and nodes in them)

» Activation functions

» Learning parameter

» Number of iterations

In particular, the architecture offers great opportunities to create powerful predictive models at a high risk of overfitting. The learning parameter controls how fast a network learns from data, but it may not suffice in preventing overfitting the training data. (See the "Looking for generalization" section of Chapter 2 for more details about why overfitting can cause problems.)

Opening the black box

You have two possible solutions to the problem of overfitting. The first is regularization, as in linear and logistic regression. You can sum all connection coefficients, squared or in absolute value, to penalize models with too many coefficients with high values (achieved by L2 regularization) or with values different from zero (achieved by L1 regularization). The second solution is also effective because it controls when overfitting happens. It's called early stop and works by checking the cost function on the validation set as the algorithm learns from the training set. (The “Learning the right direction” section of Chapter 5 provides more details about early stopping.)

You may not realize when your model starts overfitting. The cost function

calculated using the training set keeps improving as optimization progresses. However, as soon as you start recording noise from the data and stop learning general rules, you can check the cost function on an out-of-sample data (the validation sample). At some point, you’ll notice that it stops improving and starts worsening, which means that your model has reached its learning limit.

BUILDING NEURAL NETWORKS

Building a Basic Neural Network

The last chapter introduces neural networks using the simplest and most basic neural network of all: the perceptron. However, neural networks come in a number of forms, each of which has advantages. Fortunately, all the forms of neural networks follow a basic architecture and rely on certain strategies to accomplish what they need to do. If you learn how a basic neural network works, you can figure out how more complex architectures operate. The first part of this chapter discusses the basics of neural network functionality — that is, what you need to know to understand how a neural network performs useful work. It explains neural network functionality using a basic neural network that you can build from scratch using Python.

The second part of the chapter delves into some differences between neural networks. For example, you discover in Chapter 7 that individual neurons fire after reaching a particular threshold. An activation function determines when the input is sufficient for the neuron to fire, so knowing which activator functions are available is important to differentiate between neural networks. In addition, you need to know about the optimizer used to ensure that you get fast results that actually model the problem you want to solve. Finally, you need to decide how fast your neural network learns. Save yourself the time and mistakes of typing the code manually. You can find the downloadable source for this chapter in the DL4D_08_NN_From_Scratch.ipynb file. (The Introduction tells you where to download the source code for this book.)

Understanding Neural Networks

You can find many discussions about neural network architectures online (such as the one at https://www.kdnuggets.com/2018/02/8-neural-network-architectures-machine-learning-researchers-need-learn.html). The problem, however, is that they all quickly become insanely complex, making normal people want to pull out their hair. Some unwritten law seems to say that math has to become instantly abstract and so complicated that no mere mortal can understand it, but anyone can understand a neural network. Now, in this chapter you learn by putting into Python code all the essential functionalities of a neural network.

What a neural network truly represents is a kind of filter. You pour data into the top, that data percolates through the various layers you create, and an output appears at the bottom. The things that differentiate neural networks are the same sorts of things you might look for in a filter. For example, the kind of algorithm you choose determines the kind of filtering the neural network will perform. You may want to filter the lead out of the water but leave the calcium and other beneficial minerals intact, which means choosing a kind of filter to do that.

However, filters can come with controls. For example, you might choose to filter particles of one size but let particles of another size pass. The use of weights and biases in a neural network are simply a kind of control. You adjust the control to fine-tune the filtering you receive. In this case, because you’re using electrical signals modeled after those found in the brain, a signal is allowed to pass when it meets a particular condition — a threshold defined by an activation function. To keep things simple for now, though, just think about it as you would adjustments to any filter’s basic operation.

You can monitor the activity of your filter. However, unless you want to stand there all day looking at it, you probably rely on some sort of automation to ensure that the filter’s output remains constant. This is where an optimizer comes into play. By optimizing the output of the neural network, you see the results you need without constantly tuning it manually.

Finally, you want to allow a filter to work at a speed and capacity that allows it to perform its tasks correctly. Pouring water or some other substance through the filter too quickly would cause it to overflow. If you don’t pour fast enough, the filter might clog or work erratically. Adjusting the learning rate of the optimizer of a neural network enables you to ensure that the neural network produces the output you want. It’s like adjusting the pouring rate of a filter.

Neural networks can seem hard to understand. The fact that much of what they do is shrouded in mathematical complexity doesn’t help matters. However, you don’t have to be a rocket scientist to understand what neural networks are all about. All you really need to do is break them down into manageable pieces and use the right perspective to look at them. The following sections demonstrate how to code each part of a basic neural network from scratch.

Defining the basic architecture

A neural network relies on numerous computation units, the neurons, arranged into hierarchical layers. Each neuron accepts inputs from all its predecessors and provides outputs to its successors until the neural network as a whole satisfies a requirement. At this point, the network processing ends and you receive the output.

All these computations occur singularly in a neural network. The network passes over each of them using loops for loop iterations. You can also leverage the fact that most of these operations are plain multiplications, followed by addition, and take advantage of the matrix calculations shown in the “Performing matrix multiplication” section.

The example in this section creates a network with an input layer (whose dimensions are defined by the input), a hidden layer with three neurons, and a single output layer that tells whether the input is part of a class (basically a binary 0/1 answer). This architecture implies creating two sets of weights represented by two matrices (when you’re actually using matrices): » The first matrix uses a size determined by the number of inputs x 3, represents the weights that multiply the inputs, and sums them into three neurons. » The second matrix uses a size of 3 x 1, gathers all the outputs from the hidden layer, and makes that layer converge into the output.

Here’s the required Python script (which may take a while to complete running, depending on the speed of your system):

```
import numpy as np from sklearn.datasets import make_moons from
sklearn.model_selection import train_test_split import matplotlib.pyplot as
```

```
plt %matplotlib inline
```

```
def init(inp, out):    return np.random.randn(inp, out) / np.sqrt(inp)
```

```
def create_architecture(input_layer, first_layer,                        output_layer,
random_seed=0):    np.random.seed(random_seed)    layers = X.shape[1], 3 ,
1    arch = list(zip(layers[:-1], layers[1:]))    weights = [init(inp, out) for inp,
out in arch]    return weights
```

The interesting point of this initialization is that it uses a sequence of matrices to automate the network calculations. How the code initializes them matters because you can’t use numbers that are too small — there will be too little signal for the network to work. However, you must also avoid numbers that are too big because the calculations become too cumbersome to handle. Sometimes they fail, which causes the exploding gradient problem or, more often, causes saturation of the neurons, which means that you can’t correctly train a network because all the neurons are always activated.

Initializing your network using all zeros is always a bad idea because if all the neurons have the same value, they will react in the same way to the training input. No matter how many neurons the architecture contains, they operate as a single neuron.

The simpler solution is to start with initial random weights which are in the range required for the activation functions, which are the transformation functions that add flexibility to solving problems using the network. A possible simple solution is to set the weights to zero mean and one standard deviation, which in statistics is called the standard normal distribution and in the code appears as the np.random. radn command.

There are, however, smarter weight initializations for more complex networks, such as those found in this article: https://towardsdatascience.com/weight- initialization-techniques-in-neural-networks-26c649eb3b78.

Moreover, because each neuron accepts the inputs of all previous neurons, the code rescales the random normal distributed weights using the square root of the number of inputs. Consequently, the neurons and their activation functions always compute the right size for everything to work smoothly.

Documenting the essential modules

The architecture is just one part of a neural network. You can imagine it as the structure of the network. Architecture explains how the network processes data and provides results. However, for any processing to happen, you also need to code the neural network’s core functionalities.

The first building block of the network is the activation function. The example in this section provides code for the sigmoid function, one of the basic neural network activation functions. The sigmoid function is a step up from the Heaviside step function, which acts as a switch that activates at a certain threshold. A Heaviside step function outputs 1 for inputs above the threshold and 0 for inputs below it.

The sigmoid functions outputs 0 or 1, respectively, for small input values below zero or high values above zero. For input values in the range between –5 and +5, the function outputs values in the range 0–1, slowly increasing the output of released values until it reaches around 0.2 and then growing fast in a linear way until reaching 0.8. It then decreases again as the output rate approaches 1. Such behavior represents a logistic curve, which is useful for describing many natural phenomena, such as the growth of a population that starts growing slowly and then fully blossoms and develops until it slows down before hitting a resource limit (such as available living space or food).

In neural networks, the sigmoid function is particularly useful for modeling inputs that resemble probabilities, and it’s differentiable, which is a mathematical aspect that helps reverse its effects and works out the best back propagation phase mentioned in the “Going even deeper into the rabbit hole” the chapter above.

```
def sigmoid(z):    return 1/(1 + np.exp(-z))


def sigmoid_prime(s):    return s * (1 -s)
```

After you have an activation function, you can create a forward procedure, which is a matrix multiplication between the input to each layer and the weights of the connection. After completing the multiplication, the code applies the activation function to the results to transform them in a nonlinear way. The following code embeds the sigmoid function into the network’s feed-forward code. Of course, you can use any other activation function if desired. def feed_forward(X, weights):   a = X.copy()   out = list()   for W in weights: z = np.dot(a, W)   a = sigmoid(z)   out.append(a) return out

By applying the feed forward to the complete network, you finally arrive at a result in the output layer. Now you can compare the output against the real values you want the network to obtain. The accuracy function determines whether the neural network is performing predictions well by comparing the number of correct guesses to the total number of predictions provided.

def accuracy(true_label, predicted):   correct_preds = np.ravel(predicted)==true_label   return np.sum(correct_preds) / len(true_label)

The backpropagation function comes next because the network is working, but all or some of the predictions are incorrect. Correcting predictions during training enables you to create a neural network that can take on new examples and provide good predictions. The training is incorporated into its connection weights as patterns present in data that can help predict the results correctly.

To perform backpropagation, you first compute the error at the end of each layer (this architecture has two). Using this error, you multiply it by the derivative of the activation function. The result provides you with a gradient, that is, the change in weights necessary to compute predictions more correctly. The code starts by comparing the output with the correct answers (l2_error), and then computes the gradients, which are the necessary weight corrections (l2_delta). The code then proceeds to multiply the gradients by the weights the code must correct. The operation distributes the error from the output layer to the intermediate one (l1_error). A new gradient computation (l1_delta) also provides the weight corrections to apply to the input layer, which completes the process for a network with an input layer, a hidden layer, and an output layer.

def backpropagation(l1, l2, weights, y):   l2_error = y.reshape(-1, 1) - l2   l2_delta = l2_error * sigmoid_prime(l2)   l1_error = l2_delta.dot(weights[1].T)   l1_delta = l1_error * sigmoid_prime(l1)   return l2_error, l1_delta, l2_delta This is a Python code translation, in simplified form, of the formulas in Chapter 7. The cost function is the difference between the network’s output and the correct answers. The example doesn’t add biases during the feed forward phase, which reduces the complexity of the backpropagation process and makes it easier to understand.

After backpropagation assigns each connection its part of the correction that should be applied over the entire network, you adjust the initial weights to represent an updated neural network. You do so by adding to the weights of each layer, the multiplication of the input to that layer, and the delta corrections for the layer as a whole. This is a gradient descent method step in which you approach the solution by taking repeated small steps in the right direction, so you may need to adjust the step size used to solve the problem. The alpha parameters help make changing the step size possible. Using a value of 1 won’t affect the impact of the previous weight correction, but values smaller than 1 effectively reduce it.

```
def update_weights(X, l1, l1_delta, l2_delta, weights, alpha=1.0):
weights[1] = weights[1] + (alpha * l1.T.dot(l2_delta))    weights[0] =
weights[0] + (alpha * X.T.dot(l1_delta))    return weights
```

A neural network is not complete if it can only learn from data, but not predict. The last predict function pushes new data using feed forward, reads the last output layer, and transforms its values to problem predictions. Because the sigmoid activation function is so adept at modeling probability, the code uses a value halfway between 0 and 1, that is, 0.5, as the threshold for having a positive or negative output. Such a binary output could help in classifying two classes or a single class against all the others if a dataset has three or more types of outcomes to classify.

```
def predict(X, weights):    _, l2 = feed_forward(X, weights)    preds =
np.ravel((l2 > 0.5).astype(int))    return preds
```

At this point, the example has all the parts that make a neural network work. You just need a problem that demonstrates how the neural network works.

Solving a simple problem

In this section, you test the neural network code you wrote by asking it to solve a simple, but not banal, data problem. The code uses the Scikit-learn package’s make_moons function to create two interleaving circles of points shaped as two half moons. Separating these two circles requires an algorithm capable of defining a nonlinear separation function that generalizes to new cases of the same kind.

A neural network, such as the one presented earlier in the chapter, can easily handle the challenge.

```
np.random.seed(0)

coord, cl = make_moons(300, noise=0.05) X, Xt, y, yt =
train_test_split(coord,
cl,                              test_size=0.30,                              random_state=0)

plt.scatter(X[:,0], X[:,1], s=25, c=y, cmap=plt.cm.Set1) plt.show()
```

The code first sets the random seed to produce the same result anytime you want to run the example. The next step is to produce 300 data examples and split them into a train and a test dataset. (The test dataset is 30 percent of the total.) The data consists of two variables representing the x and y coordinates of points on a Cartesian graph.

Because learning in a neural network happens in successive iterations (called epochs), after creating and initializing the sets of weights, the code loops 30,000 iterations of the two half moons data (each passage is an epoch). On each iteration, the script calls some of the previously prepared core neural network functions: » Feed forward the data through the entire network. » Backpropagate the error back into the network. » Update the weights of each layer in the network, based on the back propagated error. » Compute the training and validation errors.

The following code uses comments to detail when each function operates:

```
weights = create_architecture(X, 3, 1)

for j in range(30000 + 1):

     # First, feed forward through the hidden layer    l1, l2 = feed_forward(X,
weights)         # Then, error backpropagation from output to input    l2_error,
l1_delta, l2_delta = backpropagation(l1,                                   l2, weights,
y)         # Finally, updating the weights of the network    weights =
update_weights(X, l1, l1_delta, l2_delta,                              weights,
alpha=0.05)         # From time to time, reporting the results    if (j % 5000) ==
0:        train_error = np.mean(np.abs(l2_error))        print('Epoch
{:5}'.format(j), end=' - ')        print('error:
{:0.4f}'.format(train_error),              end= ' - ')        train_accuracy =
accuracy(true_label=y,                                  predicted=(l2 >
0.5))        test_preds = predict(Xt, weights)        test_accuracy =
accuracy(true_label=yt,                                 predicted=test_preds)        print('a
train {:0.3f}'.format(train_accuracy),              end= ' | ')        print('test
{:0.3f}'.format(test_accuracy))
```

Variable j counts the iterations. At each iteration, the code tries to divide j by 5,000 and check whether the division leaves a module. When the module is zero, the code infers that 5,000 epochs have passed since the previous check, and summarizing the neural network error is possible by examining its accuracy (how many times the prediction is correct with respect to the total number of predictions) on the training set and on the test set. The accuracy on the training set shows how well the neural network is fitting the data by adapting its parameters by the backpropagation process. The accuracy on the test set provides an idea of how well the solution generalized to new data and thus whether you can reuse it.

The test accuracy should matter the most because it shows the potential usability of the neural network with other data. The training accuracy just tells you how the network scores with the present data you are using.

Looking Under the Hood of Neural Networks

After you know how neural networks basically work, you need a better understanding of what differentiates them. Beyond the different architectures, the choice of the activation functions, the optimizers and the neural network learning rate can make the difference. Knowing basic operations isn’t enough because you won’t get the results you want. Looking under the hood helps you understand how you can tune your neural network solution to model specific problems. In addition, understanding the various algorithms used to create a neural network will help you obtain better results with less effort and in a shorter time. The following sections focus on three areas of neural network differentiation.

Choosing the right activation function

An activation function simply defines when a neuron fires. Consider it a sort of tipping point: Input of a certain value won’t cause the neuron to fire because it’s not enough, but just a little more input can cause the neuron to fire. A neuron is defined in a simple manner as follows:

y = ∑ (weight * input) + bias

The output, y, can be any value between + infinity and – infinity. The problem, then, is to decide on what value of y is the firing value, which is where an activation function comes into play. The activation function determines which value is high or low enough to reflect a decision point in the neural network for a particular neuron or group of neurons.

As with everything else in neural networks, you don’t have just one activation function. You use the activation function that works best in a particular scenario. With this in mind, you can break the activation functions into these categories: » Step: A step function (also called a binary function) relies on a specific threshold for making the decision about activating or not. Using a step function means that you know which specific value will cause an activation. However, step functions are limited in that they’re either fully activated or fully deactivated —no shades of gray exist. Consequently, when attempting to determine which class is most likely correct based in a given input, a step function won’t work. » Linear: A linear function (A = cx) provides a straight-line determination of activation based on input. Using a linear function helps you determine which output to activate based on which output is most correct (as expressed by weighting). However, linear functions work only as a single layer. If you were to stack multiple linear function layers, the output would be the same as using a single layer, which defeats the purpose of using neural networks. Consequently, a linear function may appear as a single layer, but never as multiple layers. » Sigmoid: A sigmoid function (A = 1 / 1 + e-x), which produces a curve shaped like the letter C or S, is nonlinear. It begins by looking sort of like the step function, except that the values between two points actually exist on a curve, which means that you can stack sigmoid functions to perform classification with multiple outputs. The range of a sigmoid function is between 0 and 1, not – infinity to + infinity as with a linear function, so the activations are bound within a specific range. However, the sigmoid function suffers from a problem called vanishing gradient, which means that the function refuses to learn after a certain point because the propagated error shrinks to zero as it approaches far away layers. » Tanh: A tanh function (A = (2 / 1 + e-2x) – 1) is actually a scaled sigmoid function. It has a range of –1 to 1, so again, it’s a precise method for activating neurons. The big difference between sigmoid functions and tanh functions is that the tanh function gradient is stronger, which means that detecting small differences is easier, making classification more sensitive. Like the sigmoid function, tanh suffers from vanishing gradient issues. » ReLU: A ReLU, or Rectified Linear Units, function (A(x) = max(0, x)) provides an output in the range of 0 to infinity, so it's similar to the linear function except that it's also nonlinear, enabling you to stack ReLU functions. An advantage of ReLU is that it requires less processing power because fewer neurons fire. The lack of activity as the neuron approaches the 0 part of the line means that there are fewer potential outputs to look at. However, this advantage can also become a disadvantage when you have a problem called the dying ReLU. After a while, the neural network weights don't provide the desired effect any longer (it simply stops learning) and the affected neurons die — they don't respond to any input. Also, the ReLU has some variants that you should consider: » ELU (Exponential Linear Unit): Differs from ReLU when the inputs are negative. In this case, the outputs don't go to zero but instead slowly decrease to –1 exponentially. » PReLU (Parametric Rectified Linear Unit): Differs from ReLU when the inputs are negative. In this case, the output is a linear function whose parameters are learned using the same technique as any other parameter of the network. » LeakyReLU: Similar to PReLU but the parameter for the linear side is fixed.

Relying on a smart optimizer

An optimizer serves to ensure that your neural network performs fast and correctly models whatever problem you want to solve by modifying the neural network's biases and weights. It turns out that an algorithm performs this task, but you must choose the correct algorithm to obtain the results you expect. As with all neural network scenarios, you have a number of optional algorithm types from which to choose (see https://keras.io/optimizers/): » Stochastic gradient descent (SGD) » RMSProp » AdaGrad » AdaDelta » AMSGrad » Adam and its variants, Adamax and Nadam

An optimizer works by minimizing or maximizing the output of an objective function (also known as an error function) represented as E(x). This function is dependent on the model's internal learnable parameters used to calculate the target values (Y) from the predictors (X). Two internal learnable parameters are weights (W) and bias (b). The various algorithms have different methods of dealing with the objective function.

You can categorize the optimizer functions by the manner in which they deal with the derivative (dy/dx), which is the instantaneous change of y with respect to x. Here are the two levels of derivative handling: » First order: These algorithms minimize or maximize the objective function using gradient values with respect to the parameters. » Second order: These algorithms minimize or maximize the object function using the second-order derivative values with respect to the parameters. The second-order derivative can give a hint as to whether the first-order derivative is increasing or decreasing, which provides information about the curvature of the line.

You commonly use first-order optimization techniques, such as Gradient Descent, because they require fewer computations and tend to converge to a good solution relatively fast when working on large datasets.

Setting a working learning rate

Each optimizer has completely different parameters to tune. One constant is fixing the learning rate, which represents the rate at which the code updates the network's weights (such as the alpha parameter used in the example for this chapter). The learning rate can affect both the time the neural network takes to learn a good solution (the number of epochs) and the result. In fact, if the learning rate is too low, your network will take forever to learn. Setting the value too high causes instability when updating the weights, and the network won't ever converge to a good solution.

Choosing a learning rate that works is daunting because you can effectively try values in the range from 0.000001 to 100. The best value varies from optimizer to optimizer. The value you choose depends on what type of data you have. Theory can be of little help here; you have to test different combinations before finding the most suitable learning rate for training your neural network successfully.

In spite of all the math surrounding them, tuning neural networks and having them work best is mostly a matter of empirical efforts in trying different combinations of architectures and parameters.

CONVOLUTIONAL AND RECURRENT NEURAL NETWORKS

Convolutional Neural Networks

When you look inside deep learning, you may be surprised to find a lot of old technology, but amazingly, everything works as it never has before because researchers finally know how to make some simple, older solutions work together. As a result, big data can automatically filter, process, and transform data.

For instance, novel activations like Rectified Linear Units (ReLU), discussed in previous chapters, aren’t new, but you see them used in new ways. ReLU is a neural networks function that leaves positive values untouched and turns negative ones into zero; you can find a first reference to ReLU in a scientific paper by Hahnloser and others from 2000. Also, the image recognition capabilities that made deep learning so popular a few years ago aren’t new, either.

In recent years, deep learning achieved great momentum thanks to the ability to code certain properties into the architecture using Convolutional Neural Networks (CNNs), which are also called ConvNets. The French scientist Yann LeCun and other notable scientists devised the idea of CNN's at the end of the 1980s, and they fully developed their technology during the 1990s. But only now, about 25 years later, are such networks starting to deliver astonishing results, even achieving better performance than humans do in particular recognition tasks. The change has come because it’s possible to configure such networks into complex architectures that can refine their learning from lots of useful data.

CNNs have strongly fueled the recent deep learning renaissance. The following sections discuss how CNNs help in detecting image edges and shapes for tasks such as deciphering handwritten text, exactly locating a certain object in an image, or separating different parts of a complex image scene.

Save yourself the time and mistakes of typing this chapter’s example code by hand. You can find the downloadable source for this chapter in the DL4D_10_ LeNet5.ipynb file. (The Introduction tells you where to download the source code for this book.)

Beginning the CNN Tour with Character Recognition

CNNs aren’t a new idea. They appeared at the end of the 1980s as the solution for character recognition problems. Yann LeCun devised CNNs when he worked at AT&T Labs Research, together with other scientists such as Yoshua Bengio, Leon Bottou, and Patrick Haffner on a network named LeNet5. Before delving into the technology of these specialized neural networks, this chapter spends time understanding the problem of image recognition.

Digital images are everywhere today because of the pervasive presence of digital cameras, webcams, and mobile phones with cameras. Because capturing images has become so easy, a new, huge stream of data is provided by images. Being able to process images opens the doors to new applications in fields such as robotics, autonomous driving, medicine, security, and surveillance.

Understanding image basics

Processing an image for use by a computer transforms it into data. Computers send images to a monitor as a data stream composed of pixels, so computer images are best represented as a matrix of pixels values, with each position in the matrix corresponding to a point in the image.

Modern computer images represent colors using a series of 32 bits (8 bits apiece for red, blue, green, and transparency — the alpha channel). You can use just 24 bits to create a true color image, however. Computer images represent color using three overlapping matrices, each one providing information relative to one of three colors: Red, Green, or Blue (also called RGB). Blending different amounts of these three colors enables you to represent any standard human-viewable color, but not those seen by people with extraordinary perception. (Most people can see a maximum of 1,000,000 colors, which is well within the color range of the 16,777,216 colors offered by 24-bit color. Tetrachromats can see 100,000,000 colors, so you couldn’t use a computer to analyze what they see.

Generally, an image is therefore manipulated by a computer as a three-dimensional matrix consisting of height, width, and the number of channels — which is three for an RGB image, but could be just one for a black-and-white image. (Grayscale is a special sort of RGB image for which each of the three channels is the same number; see https://introcomputing.org/image-6-grayscale.html for a discussion of how conversions between color and grayscale occurs.) With a grayscale image, a single matrix can suffice by having a single number represent the 256-grayscale colors, as demonstrated by the example in Figure 10-1. In that figure, each pixel of an image of a number is quantified by its matrix values.

Given the fact that images are pixels (represented as numeric inputs), neural network practitioners initially achieved good results by connecting an image directly to a neural network. Each image pixel connected to an input node in the network. Then one or more following hidden layers completed the network, finally resulting in an output layer. The approach worked acceptably for small images and to solve small problems, giving way to different approaches for solving image recognition. As an alternative, researchers used other machine learning algorithms or figure applied intensive feature creation to transform an image into newly processed data that could help algorithms recognize the image better.

An example of image feature creation is the Histograms of Oriented Gradients (HOG), which is a computational way to detect patterns in an image and turn them into a numeric matrix. (You can explore how HOG works by viewing this tutorial from the Skimage package: http://scikit-image.org/docs/dev/auto_examples/features_ detection/plot_hog.html.)

Neural network practitioners found image feature creation to be computationally intensive and often impractical. Connecting image pixels to neurons was difficult because it required computing an incredibly large number of parameters and the network couldn’t achieve translation invariance, which is the capability to decipher a represented object under different conditions of size, distortion, or position in the image.

A neural network, which is made of dense layers as described in the previous chapters, can detect only images that are similar to those used for training — those that it has seen before — because it learns by spotting patterns at certain image locations. Also, a neural network can make many mistakes. Transforming an image before feeding it to the neural network can partially solve the problem by resizing, moving, cleaning the pixels, and creating special chunks of information for better network processing. This technique, called feature creation, requires expertise on the necessary image transformations, as well as many computations in terms of data analysis. Because of the intense level of custom work required, image recognition tasks are more the work of an artisan than a scientist. However, the amount of custom work has decreased over time as the base of libraries automating certain tasks has increased.

Explaining How Convolutions Work

Convolutions easily solve the problem of translation invariance because they offer a different image-processing approach inside the neural network. The idea started from a biological point of view by observing what happens in the human visual cortex.

A 1962 experiment by Nobel Prize winners David Hunter Hubel and Torsten Wiesel demonstrated that only certain neurons activate in the brain when the eye sees certain patterns, such as horizontal, vertical, or diagonal edges. In addition, the two scientists found that the neurons organize vertically, in a hierarchy, suggesting that visual perception relies on the organized contribution of many single, specialized neurons. (You can find out more about this experiment by reading the article at https://knowingneurons.com/2014/10/29/hubel-and-wiesel-the- neural-basis-of-visual-perception/.) Convolutions simply take this idea and, by using mathematics, apply it to image processing in order to enhance the capabilities of a neural network to recognize different images accurately.

Understanding convolutions

To understand how convolutions work, you start from the input. The input is an image composed of one or more pixel layers, called channels, and the image uses values from 0, which means that the individual pixel is fully switched off, to 255, which means that the individual pixel is switched on. (Usually, the values are stored as integers to save memory.) As mentioned in the preceding section of this chapter, RGB images have individual channels for red, green, and blue colors. Mixing these channels generates the palette of colors as you see them on the screen.

A convolution works by operating on small image chunks across all image channels simultaneously. (Picture a slice of layer cake, with, each piece showing all the layers). Image chunks are simply a moving image window: The convolution window can be a square or a rectangle, and it starts from the upper left of the image and moves from left to right and from top to bottom. The complete tour of the window over the image is called a filter and implies a complete transformation of the image. Also important to note is that when a new chunk is framed by the window, the window then shifts a certain number of pixels; the amount of the slide is called a stride. A stride of 1 means that the window is moving one pixel toward right or bottom; a stride of 2 implies a movement of two pixels; and so on.

Every time the convolution window moves to a new position, a filtering process occurs to create part of the filter described in the previous paragraph. In this process, the values in the convolution window are multiplied by the values in the kernel (a small matrix used for blurring, sharpening, embossing, edge detection, and more — you choose the kernel you need for the task in question). (The article at http://setosa.io/ev/image-kernels/ tells you more about various kernel types.) The kernel is the same size as the convolution window. Multiplying each part of the image with the kernel creates a new value for each pixel, which in a sense is a new processed feature of the image. The convolution outputs the pixel value and when the sliding window has completed its tour across the image, you have filtered the image.

As a result of the convolution, you find a new image having the following characteristics:

» If you use a single filtering process, the result is a transformed image of a single channel.

» If you use multiple kernels, the new image has as many channels as the number of filters, each one containing specially processed new feature values. The number of filters is the filter depth of a convolution.

» If you use a stride of 1, you get an image of the same dimensions as the original.

» If you use strides of a size above 1, the resulting convoluted image is smaller than the original (a stride of size two implies halving the image size).

» The resulting image may be smaller depending on the kernel size, because the kernel has to start and finish its tour on the image borders. When processing the image, a kernel will eat up its size minus one. For instance, a kernel of 3 x 3 pixels processing a 7-x-7-pixel image will eat up 2 pixels from the height and width of the image, and the result of the convolution will be an output of size 5 x 5 pixels. You have the option to pad the image with zeros at the border (meaning, in essence, to put a black border on the image) so that the convolution process won’t reduce the final output size. This strategy is called same padding. If you just let the kernel reduce the size of your starting image, it’s called valid padding.

Image processing has relied on the convolution process for a long time. Convolution filters can detect an edge or enhance certain characteristics of an image.

The problem with using convolutions is that they are human made and require effort to figure out. When using a neural network convolution instead, you just set the following:

» The number of filters (the number of kernels operating on an image that is its output channels)

» The kernel size (set just one side for a square; set width and height for a rectangle) » The strides (usually 1- or 2-pixel steps)

» Whether you want the image black bordered (choose valid padding or same padding)

After determining the image-processing parameters, the optimization process determines the kernel values used to process the image in a way to allow the best classification of the final output layer. Each kernel matrix element is therefore a neural network neuron and modified during training using backpropagation for the best performance of the network itself.

Another interesting aspect of this process is that each kernel specializes in finding specific aspects of an image. For example, a kernel specialized in filtering features typical of cats can find a cat no matter where it is in an image and, if you use enough kernels, every possible variant of an image of a kind (resized, rotated, translated) is detected, rendering your neural network an efficient tool for image classification and recognition. borders of an image are easily detected after a 3-x-3-pixel kernel is applied. This kernel specializes in finding edges, but another kernel could spot different image features. By changing the values in the kernel, as the neural network does during backpropagation, the network finds the best way to process images for its regression or classification purpose.

The kernel is a matrix whose values are defined by the neural network optimization, multiplied by a small patch of the same size moving across the image, but it can be intended as a neural layer whose weights are shared across the different input neurons. You can see the patch as an immobile neural layer connected to the many parts of the image always using the same set of weights. It is exactly the same result.

Keras offers a convolutional layer, Conv2D, out of the box. This Keras layer can take both the input directly from the image (in a tuple, you have to set the input_ shape the width, height, and number of channels of your image) or from another layer (such as another convolution). You can also set filters, kernel_size, strides, and padding, which are the basic parameters for any convolutional layers, as described earlier in the chapter.

When setting a Conv2D layer, you may also set many other parameters, which are actually a bit too technical and maybe not necessary for your first experiments with CNNs. The only other parameters you may find useful now are activation, which can add an activation of your choice, and name, which sets a name for the layer. Simplifying the use of pooling

Convolutional layers transform the original image using various kinds of filtering. Each layer finds specific patterns in the image (particular sets of shapes and colors that make the image recognizable). As this process continues, the complexity of the neural network grows because the number of parameters grows as the network gains more filters. To keep the complexity manageable, you need to speed the filtering and reduce the number of operations.

Pooling layers can simplify the output received from convolutional layers, thus reducing the number of successive operations performed and using fewer convolutional operations to perform filtering. Working in a fashion similar to convolutions (using a window size for the filter and a stride to slide it), pooling layers operate on patches of the input they receive and reduce a patch to a single number, thus effectively downsizing the data flowing through the neural network.

Figure 10-5 represents the operations done by a pooling layer, receiving as input the filtered data represented by the left 4-x-4 matrix: operating on it using a window of size 2 pixels and moving by a stride of 2 pixels. As a result, the pooling layer produces the right output: a 2-x-2 matrix. The network applies the pooling operation on four patches represented by four different colored parts of the matrix. For each patch, the pooling layer computes the maximum value and saves it as an output.

The current example relies on the max pooling layer because it uses the max transformation on its sliding window. You actually have access to four principal types of pooling layers: » Max pooling » Average pooling » Global max pooling » Global average pooling

In addition, these four pooling layer types have different versions, depending on the dimensionality of the input they can process: » 1-D pooling: Works on vectors. Thus, 1-D pooling is ideal for sequence data such as temporal data (data representing events following each other in time) or text (represented as sequences of letters or words). It takes the maximum or the average of contiguous parts of the sequence. » 2-D pooling: Fits spatial data that fits a matrix. You can use 2-D pooling for a grayscale image or each channel of an RBG image separately. It takes the maximum or the average of small patches (squares) of the data. » 3-D pooling: Fits spatial data represented as spatial-temporal data. You could use 3-D pooling for images taken across time. A typical example is to use magnetic resonance imagining (MRI) for a medical examination. Radiologists use an MRI to examine body tissues with magnetic fields and radio waves. (See the article from Stanford AI for healthcare to learn more about the contribution of deep learning: https://medium.com/stanford-ai-for- healthcare/dont-just-scan-this-deep-learning-techniques-for- mri-52610e9b7a85.) This kind of pooling takes the maximum or the average of small chunks (cubes) from the data.

You can find all these layers described in the Keras documentation, together with all their parameters, at https://keras.io/layers/pooling/.

Describing the LeNet architecture

You may have been amazed by the description of a CNN in the preceding section, and about how its layers (convolutions and max pooling) work, but you may be even more amazed at discovering that it’s not a new technology; instead, it appeared in the 1990s. The following sections describe the LeNet architecture in more detail.

Considering the underlying functionality The key person behind this innovation was Yann LeCun, who was working at AT&T Labs Research as head of the Image Processing Research Department. LeCun specialized in optical character recognition and computer vision. Yann LeCun is a French computer scientist who created convolutional neural networks with Léon Bottou, Yoshua Bengio, and Patrick Haffner. At present, he is the Chief AI

Scientist at Facebook AI Research (FAIR) and a Silver Professor at New York University (mainly affiliated with the NYU Center for Data Science). His personal home page is at http://yann.lecun.com/.

In the late 1990s, AT&T implemented LeCun LeNet 5 to read ZIP codes for the United States Postal Service. The company also used LeNet5 for ATM check readers, which can automatically read the check amount. The system doesn’t fail, as reported by LeCunn at https://pafnuty.wordpress.com/2009/06/13/ yann-lecun/. However, the success of the LeNet passed almost unnoticed at the time because the AI sector was undergoing an AI winter: both the public and investors were significantly less interested and attentive to improvements in neural technology than they are now.

Part of the reason for an AI winter is that many researchers and investors lost their faith in the idea that neural networks would revolutionize AI. Data of the time lacked the complexity for such a network to perform well. (ATMs and the USPS were notable exceptions because of the quantities of data they handled.) With a lack of data, convolutions only marginally outperform regular neural networks made of connected layers. In addition, many researchers achieved results comparable to LeNet5 using brand-new machine learning algorithms such as Support Vector Machines (SVMs) and Random Forests, which were algorithms based on mathematical principles different from those used for neural networks.

You can see the network in action at http://yann.lecun.com/exdb/lenet/ or in this video, in which a younger LeCun demonstrates an earlier version of the network: https://www.youtube.com/watch?v=FwFduRA_L6Q. At that time, having a machine able to decipher both typewritten and handwritten numbers was quite a feat.

The LeNet 5 architecture consists of two sequences of convolutional and average pooling layers that perform image processing. The last layer of the sequence is then flattened; that is, each neuron in the resulting series of convoluted 2-D arrays is copied into a single line of neurons. At this point, two fully connected layers and a softmax classifier complete the network and provide the output in terms of probability. The LeNet5 network is really the basis of all the CNN's that follow. Recreating the architecture using Keras will explain it layer-by- layer and demonstrate how to build your own convolutional networks.

Detecting Edges and Shapes from Images

Convolutions process images automatically and perform better than a densely connected layer because they learn image patterns at a local level and can retrace them in any other part of the image (a characteristic called translation invariance). On the other hand, traditional dense neural layers can determine the overall

characteristics of an image in a rigid way without the benefit of translation invariance. It’s like the difference between learning a book by memorizing the text in meaningful chunks or memorizing it word by word. The student (the convolutions) who learned chunk by chunk can better abstract the book content and is ready to apply that knowledge to similar cases. The student (the dense layer) who learned it word by word struggles to extract something useful.

CNNs are not magic, nor are they a black box. You can understand them through image processing and leverage their functionality to extend their capabilities to new problems. This feature helps solve a series of computer vision problems that data scientists deemed too hard to crack using older strategies.

Visualizing convolutions

A CNN uses different layers to perform specific tasks in a hierarchical way. Yann LeCun (see the “Beginning the CNN Tour with Character Recognition” section, early in this chapter) noticed how LeNet first processed edges and contours, and then motifs, and then categories, and finally objects. Recent studies further unveil how convolutions really work: » Initial layers: Discover the image edges » Middle layers: Detect complex shapes (created by edges) » Final layers: Uncover distinctive image features characteristic of the image type that you want the network to classify (for instance, the nose of a dog or the ears of a cat)

This hierarchy of patterns discovered by convolutions also explains why deep convolutional networks perform better than shallow ones: the more stacked convolutions there are, the better the network can learn more and more complex and useful patterns for successful image recognition. The image of a dog is processed by convolutions, and the first layer grasps patterns. The second layer accepts these patterns and assembles them into a cat. If the patterns processed by the first layer seem too general to be of any use, the patterns unveiled by the second layer recreate more characteristic dog features that provide an advantage to the neural network in recognizing dogs.

Recurrent Neural Networks

This part explores how deep learning can deal with information that flows. Reality is not simply changeable, but is changeable in a progressive way that is made predictable by observing the past. If a picture is a static snapshot of a moment in time, a video, consisting of a sequence of related images, is flowing information, and a film can tell you much more than a single photo or a series of photos can. Likewise for short and long textual data (from tweets to entire documents or books) and for all numeric series that represent something occurring along a timeline (for instance, a series the about sales of a product or the quality of the air by day in a city).

This part explains a series of new layers, the recurrent networks, and all their improvements, such as the LSTM and GRU layers. These technologies are behind the most astonishing deep learning applications that you can experiment with today. You commonly see them used on your mobile phone or at home. For example, you use this kind of application when chatting with smart speakers such as Siri, Google Home, or Alexa. Another application is translating your conversation into another language using Google Translate. Behind each of these technologies are a distinctive neural architecture and application-specific data used for training — some public and some proprietary. Even with these differences in data source and technique, the layers that make everything possible are precisely the same layers that you import from TensorFlow and Keras and use when coding your applications.

Introducing Recurrent Networks

Neural networks provide a transformation of your input into a desired output. Even in deep learning, the process is the same, although the transformation is more complex. In contrast to a simpler neural network made up of few layers, deep learning relies on more layers to perform complex transformations. The output from a data source connects to the input layer of the neural network, and the input layer starts processing the data. The hidden layers map the patterns and relate them to a specific output, which could be a value or a probability. This process works perfectly for any kind of input, and it works especially well for images.

After each layer processes its data, it outputs the transformed data to the next layer. That next layer processes the data with complete independence from the previous layers. The use of this strategy implies that if you are feeding a video to your neural network, the network will process each image singularly, one after the other, and the result won’t change at all even if you shuffled the order of the provided images. When running a network in such a fashion, using the architectures described in previous chapters of this book, you won’t get any advantage from the order of the information processing.

However, experience also teaches that to understand a process, you sometimes have to observe events in sequence. When you use the experience gained from a previous step to explore a new step, you can reduce the learning curve and lessen the time and effort needed to understand each step.

Modeling sequences using memory

The kind of neural architectures seen so far don’t allow you to process a sequence of elements simultaneously using a single input. For instance, when you have a series of monthly product sales, you accommodate the sales figures using twelve inputs, one for each month, and let the neural network analyze them at one time. It follows that when you have longer sequences, you need to accommodate them using a larger number of inputs, and your network becomes quite huge because each input should connect with every other input. You end up having a network characterized by a large number of connections (which translates into many weights), too.

Recurrent Neural Networks (RNNs) are an alternative to the solutions found in previous chapters, such as the perceptron and CNNs. They first appeared in the 1980s, and various researchers have worked to improve them until they recently gained popularity thanks to the developments in deep learning and computational power. The idea behind RNNs is simple, they examine each element of the sequence once and retain memory of it so they can reuse it when examining the next element in the sequence. It’s akin to how the human mind works when reading text: a person reads letter by letter the text but understands words by remembering each letter in the word. In a similar fashion, an RNN can associate a word to a result by remembering the sequence of letters it receives. An extension of this technique makes it possible ask an RNN to determine whether a phrase is positive or negative—a widely used analysis called sentiment analysis. The network connects a positive or negative answer to certain word sequences it has seen in training examples.

You represent an RNN graphically as a neural unit (also known as a cell) that connects an input to an output but also connects to itself. This self-connection represents the concept of recursion, which is a function applied to itself until it achieves a particular output. One of the most commonly used examples of recursion is computing a factorial, as described at https://www. geeksforgeeks.org/recursion/. The figure shows a specific RNN example using a letter sequence to make the word jazz. The right side of the figure depicts a representation of the RNN unit behavior receiving jazz as an input, but there is actually only the one unit, as shown on the left.

1-1 shows a recursive cell on the left and expands it as an unfolded series of units that receives the single letters of the word jazz on the right. It starts with j, followed by the other letters. As this process occurs, the RNN emits an output and modifies its internal parameters. By modifying its internal parameters, the unit learns from the data it receives and from the memory of the previous data. The sum of this learning is the state of the RNN cell. When discussing neural networks in previous chapters, this book talks solely about weights. With RNNs, you also need to know the term state. The weights help process the input into an output in an RNN, but the state contains the traces of the information the RNN has seen so far, so the state affects the functioning of the RNN. The state is a kind of short-term memory that resets after a sequence completes. As an RNN cell gets pieces of a sequence, it does the following: 1. Processes them, changing the state with each input. 2. Emits an output. 3. After seeing the last output, the RNN learns the best weights for mapping the input into the correct output using backpropagation.

Recognizing and translating speech

The capability to recognize and translate between languages becomes more important each day as economies everywhere become increasingly globalized. Language translation is an area in which AI has a definite advantage over humans — so much so that articles like https://www.digitalistmag.com/digital-economy/ 2018/07/06/artificial-intelligence-is-changing-translation-industry- but-will-it-work-06178661 and https://www.forbes.com/sites/bernardmarr/ 2018/08/24/will-machine-learning-ai-make-human-translators-an- endangered-species/#535ec9703902 are beginning to question how long the human translator will remain viable.

Of course, you must make the translation process viable using deep learning. From a neural architecture perspective, you have a couple of choices: » Keep all the outputs provided by the RNN cell » Keep the last RNN cell output

The last output is the output of the entire RNN because it's produced after completing the sequence examination. However, you can use the previous outputs if you need to predict another sequence or you intend to stack more RNN cells after the current one, such as when working with Convolutional Neural Networks (CNNs). Staking RNNs vertically enables the network to learn complex sequence patterns and become more effective in producing predictions.

You can also stack RNNs horizontally in the same layer. Allowing multiple RNNs to learn from a sequence can help it get more from the data. Using multiple RNNs is similar to CNNs, in which each single layer uses depths of convolutions to learn details and patterns from the image. In the multiple RNNs case, a layer can grasp different nuances of the sequence they are examining. Designing grids of RNNs, both horizontally and vertically, improves predictive performances. However, deciding how to use the output determines what a deep learning architecture powered by RNNs can achieve. The key is the number of elements used as inputs and the sequence length expected as output. As the deep learning network synchronizes the RNN outputs, you get your desired outcome.

You have a few possibilities when using multiple RNNs:

» One to one: When you have one input and expect one output. The examples in this book so far use this approach. They take one case, made up of a certain number of informative variables, and provide an estimate, such as a number or probability.

» One to many: Here you have one input and you expect a sequence of outputs as a result. Automatic captioning neural networks use this approach: You input a single image and produce a phrase describing image content.

» Many to one: The classic example for RNNs. For example, you input a textual sequence and expect a single result as output. You see this approach used for producing a sentiment analysis estimate or another classification of the text.

» Many to many: You provide a sequence as input and expect a resulting sequence as output. This is the core architecture for many of the most impressive deep learning–powered AI applications. This approach is used for machine translation (such as a network that can automatically translate a phrase from English to German), chatbots (a neural network that can answer your questions and argue with you), and sequence labeling (classifying each of the images in a video).

Machine translation is the capability of a machine to translate, correctly and meaningfully, one human language into another one. This capability is something that scientists have striven to achieve for long time, especially for military purposes. You can read the fascinating story of all the attempts to perform machine

FIGURE translation by U.S. and Russian scientists in the article at http://vas3k.com/ blog/machine_translation/ by Vasily Zubarev. The real breakthrough happened only after Google launched its Google Neural Machine Translation (GNMT), which you can read more about on the Google AI blog: https://ai.googleblog. com/2016/09/a-neural-network-for-machine.html. GNMT relies on a series of RNNs (using the many-to-many paradigm) to read the word sequence in the language you want to translate from (called the encoder layer) and return the results to another RNN layer (the decoder layer) that transforms it into translated output.

Neural machine translation needs two layers because the grammar and syntax of one language can be different from another. A single RNN can’t grasp two language systems at the same time, so the encoder-decoder couple is needed to handle the two languages. The system isn’t perfect, but it’s an incredible leap forward from the previous solutions described in Vasily Zubarev’s article, greatly reducing errors in word order, lexical mistakes (the chosen translation word), and grammar (how words are used).

Moreover, performance depends on the training set, the differences between the languages involved, and their specific characteristics. For instance, because of how sentence structure is built in Japanese, the Japanese government is now investing in a real-time voice translator to help during the Tokyo Olympic

Games in 2020 and to boost tourism by developing an advanced neural network solution (see https://www.japantimes.co.jp/news/2015/03/31/reference/ translation-tech-gets-olympic-push/ for details).

RNNs are the reason your voice assistant can answer you or your automatic translator can give you a foreign language translation. Because an RNN is simply a recurring operation of multiplication and summation, deep learning networks can’t really understand any meaning; they simply process words and phrases based on what they learned during training.

Placing the correct caption on pictures

Another possible application of RNNs using the many-to-many approach is caption generation, which involves providing an image to a neural network and receiving a text description that explains what’s happening in the image. In contrast to chatbots and machine translators, whose output is consumed by humans, caption generation works with robotics. It does more than simply generate image or video descriptions. Caption generation can help people with impaired vision perceive their environment using devices like the Horus wearable (https://horus.tech/ horus/?l=en_us) or build a bridge between images and knowledge bases (which are text based) for robots — allowing them to understand their surroundings better. You start from specially devised datasets such as the Pascal Sentence Dataset (see it at http://vision.cs.uiuc.edu/pascal-sentences/); the Flickr 30K (http://shannon.cs.illinois.edu/DenotationGraph/), which consists of Flickr images annotated by crowd sourcing; or the MS Coco dataset (http:// cocodataset.org). In all these datasets, each image includes one or more phrases explaining the image content. For example, in the MS Coco dataset sample number 5947 (http://cocodataset.org/#explore?id=5947) you see four flying airplanes that you could correctly caption as: » Four airplanes in the sky overhead on an overcast day » Four single-engine planes in the air on a cloudy day » A group of four planes flying in formation » A group of airplanes flying through the sky » A fleet of planes flying through the sky

A well-trained neural network should be able to produce analogous phrases, if presented with a similar photo. Google first published a paper on the solution for this problem, named the Show and Tell network or Neural Image Caption (NIC), in 2014, and then updated it one year later (see the article at https://arxiv.org/ pdf/1411.4555.pdf).

Google has since open sourced the NIC and offered it as part of the

TensorFlow framework. As a neural network, it consists of a pretrained CNN (such as Google LeNet, the 2014 winner of the ImageNet competition; see the “Describing the LeNet architecture” section of Chapter 10 for details) that processes images similarly to transfer learning. An image is turned into a sequence of values representing the high-level image features detected by the CNN. During training, the embedded image passes to a layer of RNNs that memorize the image characteristics in their internal state. The CNN compares the results produced by the RNNs to all the possible descriptions provided for the training image and an error is computed. The error then back propagated to the RNN’s part of the network to adjust the RNN’s weights and help it learn how to caption images correctly. After repeating this process many times using different images, the network is ready to see new images and provide its description of these new images.

Explaining Long Short-Term Memory

The use of short-term memory in RNNs may seem to be able to solve every possible deep learning problem. However, RNNs don’t come entirely without flaws. The problem with RNNs arises from their key characteristic, which is the recursion of the same information over time. The same information, passing many times through the same cells, can become progressively dampened and then disappear if the cell weights are too small. This is the so-called vanishing gradient problem, when a back propagated error-correcting signal disappears when passed through a neural network. Because of the vanishing gradient problem, you can’t stack too many layers of RNNs or updating them becomes difficult.

RNNs experience problems that are even more difficult. In backpropagation, the gradient (a correction) deals with the error correction that the networks produce when predicting. The layers before the prediction distribute the gradient to the input layers, and they provide the correct weight update. Layers reached by a small gradient update effectively stop learning.

In fact, the internally back propagated signals of RNNs tend to disappear after a few recursions, so the sequences that the neural network updates and learns better are the most recent ones. The network forgets early signals and can’t relate previously seen signals to more recent input. An RNN, therefore, can easily become too short sighted, and you can’t successfully apply it to problems that require a longer memory.

Backpropagation in an RNN layer operates both through the layer toward other layers and internally, inside each RNN cell, adjusting its memory. Unfortunately, no matter how strong the signal is, after a while the gradient dampens and vanishes.

Short memory and the vanishing gradient make it hard for RNNs to learn longer sequences. Applications like image captioning or machine translation need a keen memory on all the parts of the sequence. Consequently, most applications require an alternative, and basic RNNs have been replaced by different recurrent cells.

Defining memory differences

Two scientists studied the vanishing gradient problem in RNNs and published a milestone paper in 1997 that proposed a solution for RNNs. Sepp Hochreiter, a computer scientist who made many contributions to the fields of machine learning, deep learning, and bioinformatics, and Jürgen Schmidhuber, a pioneer in the field of artificial intelligence, published “Long Short-Term Memory” in the MIT Press Journal Neural Computation. (http://www.bioinf.at/publications/older/2604. pdf). The article introduced a new recurrent cell concept that now serves as the foundation of all the incredible deep learning applications using sequences. Originally refused because it was too innovative (ahead of its time), the new cell concept proposed by the article, named LSTM (short for long short-term memory) is used today to perform more than 4 billion neural operations per day, according to Schmidhuber’s personal home page (http://people.idsia.ch/~juergen/). LTSM is considered the standard for machine translation and chatbots. Google, Apple, Facebook, Microsoft, and Amazon have all developed products around the LSTM technology devised by Hochreiter and Schmidhuber. Products such as smart voice assistants and machine translators would work differently if LSTM were not invented.

The core idea behind LSTM is for the RNN to discriminate the state between short and long term. The state is the memory of the cell, and LSTM separates into different channels: » Short term: Input data directly mixes with data arriving from the sequence » Long term: Picks up from short-term memory only the elements that need to be retained for a long time

Moreover, the channel for long-term memory has fewer parameters to tune. Long-term memory uses some additions and multiplications with the elements arriving from the short-term memory and nothing more, making it an almost direct information highway. (The vanishing gradient can't stop the flow of information.)

Walking through the LSTM architecture

LSTMs are arranged around gates, which are internal mechanisms that use summation, multiplication, and an activation function to regulate the flow of information inside the LSTM cell. By regulating the flow, a gate can maintain, enhance, or discard the information that has arrived from a sequence in both short- and long-term memory. This flow is reminiscent of an electric circuit.

The different roots and gates may seem a bit complicated at first, but the following sequence of steps helps you understand them:

1. The short-term memory arriving from a previous state (or from random values) meets the newly inputted part of the sequence and they mix together, creating a first derivation.

2. The short-term memory signal, carrying both the exiting signal and the newly inputted signal, tries to reach the long-term memory by passing through the forget gate, which is used to forget certain data. (Technically, you see branching where the signal is duplicated.)

3. The forget gate decides what short-term information it should discard before passing it to the long-term memory. A sigmoid activation cancels the signals that aren't useful and enhances what instead seems important to keep and remember.

4. Information passing through the forget gate arrives at the long-term memory channel carrying the information from the previous states.

5. The values of the long-term memory and the output from the forget gate are multiplied together.

6. The short-term memory that didn't pass through the forget gate is duplicated again and takes another branch; that is, one part proceeds to the output gate and the other one faces the input gate.

7. At the input gate, the short-term memory data passes separately through a sigmoid function and a tanh function. The outputs of these two functions are then first multiplied together, and then added to the long-term memory. The

effect on long-term memory depends on the sigmoid, which acts in order to forget or remember if the signal is deemed important.

8. After the addition with the outputs from the input gate, the long-term memory doesn’t change. Being made of selected inputs from short-term memory, the long-term memory carries information longer in the sequence and doesn’t react to a temporal gap between.

9. Long-term memory provides information directly to the next state. It’s also sent to the output gate, where the short-term memory also converges. This last gate normalizes the data from long-term memory using tanh activation and filters the short-term memory using the sigmoid function. The two results are multiplied by one another and then sent to the next state.

LSTMs use both sigmoid and tanh activations for their gates. The principle to remember is that a tanh function normalizes its input between –1 and 1, and a sigmoid function reduces it between 0 and 1. Therefore, whereas a tanh activation function keeps the input between a workable range of values, a sigmoid can switch the input off because it pushes weaker signals toward zero, extinguishing them. In other words, the sigmoid function helps remembering (enhancing the signal) and forgetting (dampening the signal).

Discovering interesting variants

LSTM has some variants, all called with additional numbers or letters in the name, such as LSTM 4, LSTM4a, LSTM5, LSTM5a, and LSMT6, to show that they have a modified architecture although the core concepts of the solution remain. A popular and relevant modification that you find in these variants is the use of peephole connections, which are simply data pipelines that allow all or some of the gate layers to look at the long-term memory (in RNN terms, the cell’s state). By allowing peeping at long-term memory, the RNN can base decisions for the short term on previously seen patterns that were consolidated in the run. On Keras, you can find the regular LSTM implementation using the keras.layers.LSTM command (keras.layers.CuDNNLST is the GPU version), which suffices for most applications. If you need to test the peephole variants, you can explore the TensorFlow implementation (see https://www.tensorflow.org/api_docs/python/tf/nn/ rnn_cell/LSTMCell) that offers more options at the architecture level of the LSTM cell.

Another variant is more radical. The Gated Recurrent Units (also known as GRUs) first appeared in the paper called "Learning Phrase Representations using RNN Encoder-Decoder for Statistical Machine Translation" at https://arxiv.org/ pdf/1406.1078.pdf. GRUs act as a simplification of the LSTM architecture. In fact, they operate using information gates whose parameters are learnable in the same fashion as LSTM. Overall, the flow of information in a GRU cell takes a linear route because GRU uses only a working memory (equivalent to the long-term memory in LSTM terms). This working memory is refreshed by an update gate using the present information provided to the network. The updated information is then summed again with the original working memory in a gate combining the two, which is called a reset gate because it selects the working memory information to effectively retain a memory of the data being released to the next sequence step. You can see a simple schema of the flow in Figure 11-4.

Contrary to the LSTM, GRUs use a reset gate that stops the information that should be forgotten. GRUs also use an update gate that maintains the useful signals. GRUs have a unique memory, with no distinction between a long and short one.

You can use both GRUs and LSTM layers in your networks without changing the code too much. Import the layer using keras.layers.GRU (or keras.layers. CuDNNGRU for the GPU-only version that relies on the NVIDIA CuDNN library; see https://developer.nvidia.com/cudnn for details) and interact with it as an LSTM layer. You specify the parameter units by defining the number of GRU units needed in one layer. Switching from LSTM to GRU provides these advantages as well as some trade-offs: » GRUs treat the signals as LSTMs do and potentially avoid the vanishing gradient problem, but they don't distinguish between long and short memory because they rely on a single working memory — a cell state processed repeatedly through a GRU cell. » GRUs are less complex than LSTMs, but they are also less capable of remembering past signals, thus LSTMs have an advantage when dealing with longer sequences. » GRUs train faster than LSTMs (they have fewer parameters to adjust). » GRUs perform better than LSTMs when you have less training data, because they are less likely to overfit the information they receive.

Getting the necessary attention

When reading about LSTM and GRU layers applied to language problems, frequently you find the attention mechanism mentioned as the most effective way to solve complex problems, such as » Asking a neural network answer questions » Classifying phrases » Translating a text from one language into another The attention mechanism is considered the state-of-the-art solution for solving these complex problems and, in spite of being absent from presently available layers in the TensorFlow and Keras packages, finding a working open source implementation of it or even programming one yourself isn't difficult.

You can start to create your own attention mechanism by looking at the open source implementation developed by Philippe Rémy, a research engineer, at https://github.com/philipperemy/keras-attention-mechanism.

First exposed in the paper "Neural machine translation by jointly learning to align and translate," by Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio in 2014 (https://arxiv.org/abs/1409.0473v7), attention layers that implement an attention mechanism are vectors of weights expressing the importance of an element in a set processed by a deep neural network. Often the set of elements includes a sequence processed by RNNs, but it could also be an image. In fact, an attention layer can solve two kinds of problems: » When processing long sequences of words, related words might appear far apart in the sequence. For instance, pronouns are typically difficult for RNNs to handle because they can't relate the pronoun to elements passed previously in the sequence. An attention layer can highlight key elements in a phrase before the RNN starts processing the sequence. » When processing large images, many objects appearing in the picture can distract the neural network from learning how to classify target objects correctly. An example is when building a network to recognize landmarks in holiday photos. An attention layer can detect what portion of the photo the neural network should process and suggest that the RNN ignore irrelevant elements such as a person, dog, or car present in the picture.

In a neural network, the attention layer is usually placed following a recurrent layer such as an LSTM or a GRU. In 2017, researchers from Google created a standalone attention mechanism that can work without relying on previous recurrent layers and that performs much better than previous solutions. They called such architecture a Transformer.

THE FUTURE OF DEEP LEARNING

Deep learning (DL) became an overnight “star” when a robot player beat a human player in the famed game of AlphaGo. Deep learning training and learning methods have been widely acknowledged for “humanizing” machines. Many of the advanced automation capabilities now found in enterprise AI platforms are due to the rapid growth of machine learning (ML) and deep learning technologies. *A Deep Dive into Deep Learning in 2019* comments on the “ubiquitous” presence of DL in many facets of AI — be it NLP or computer vision applications. Gradually, AI and DL-enabled automated systems, tools, and solutions are penetrating and taking over all business sectors —from marketing to customer experience, from virtual reality to natural language processing (NLP) — the digital impact is everywhere.

The Future of Deep Learning

Predictions for the Future of Deep Learning claims that in the next 5 to 10 years, DL will be democratized via every software-development platform. DL tools will become a standard part of the developer’s toolkit. Reusable DL components, incorporated into standard DL libraries, will carry the training characteristics of its previous models to speed up learning. As automation of deep learning tools continue, there’s an inherent risk the technology will develop into something so complex that the average developer will find themselves totally ignorant.

Predictions About Deep Learning

Deep Learning Future Trends in a Nutshell

Some of the primary trends that are moving deep learning into the future are:

- Current growth of DL research and industry applications demonstrate its “ubiquitous” presence in every facet of AI — be it NLP or computer vision applications.
- With time and research opportunities, unsupervised learning methods may deliver models that will closely mimic human behavior.
- The apparent conflict between consumer data protection laws and research needs of high volumes of consumer data will continue.

- Deep learning technology’s limitations in being able to “reason” is a hindrance to automated, decision-support tools.
- Google’s acquisition of DeepMind Technologies holds promise for global marketers.
- The future ML and DL technologies must demonstrate learning from limited training materials, and transfer learning between contexts, continuous learning, and adaptive capabilities to remain useful.
- Though globally popular, deep learning may not be the only savior of AI solutions.
- If deep learning technology research progresses in the current pace, developers may soon find themselves outpaced and will be forced to take intensive training.

CONCLUSION

The purpose of this book is to prepare ourselves for real-world machine learning problems. We began with the introduction of deep and ended it with the future of deep learning. We’ve discussed in depth the typical tasks, common challenges, and best practices for each of these four stages. Practice makes perfect. The most important best practice is practice itself. Get started with a real-world project to deepen your understanding and apply what we have learned throughout the entire book.